CHAPTER II

BACKGROUND

2.0 Background

This chapter describes background relating to weak methods, strong methods and evolutionary algorithms. It is argued that evolutionary algorithms consist a new type of weak method, called the evolutionary weak method, that adapts its task-independent method of search based on information garnered in previous portions of the problem space. First, this chapter begins with a general discussion of search and intelligence.

2.1 Search and Intelligence

Search has always been an integral part of machine intelligence from the first computer chess programs to the current surge in learning methods. In general, search is used in two distinct ways. In a standard search, an object with specific properties is to be located from a known set of objects. For problem solving search, an initial object, a goal object and a set of operators is provided with the objective being the identification of an ordered list of operators that transform the initial object into the goal object. Problem solving search can be equated as a standard search merely by realizing that the set of objects to be searched is all possible orderings of the operators.

Let the set of possible search objects be denoted by S. S, in and of itself, is unorganized and can contain any form of object from numbers or mathematical equations to computer programs. To organize S, a set of operators, O, can be defined so that $o_i \in O, o_i: S \rightarrow S$. These operators set up an ordering over the set of objects, usually called the *search space* or *problem space*. Selecting a set of beneficial operators is typically done with some understanding of the task and intended representation of the objects.

Often, the search space is described as a Cartesian space, with each point representing a single member of the set of objects and each operator of O denoting a dimension orthogonal to the rest. Thus if $|O| = n$, the search space is an n-dimensional space. At times the connectivity of the search space is described as a graph or tree with operators serving as edges between nodes which represent the various objects. In either case, the *distance* or *depth* between objects is the number of operator applications required to transform one search space object into the other. In the event that more than one path through the search space connect two objects, their distance is the minimum path distance.

Next, an *evaluation function*, f: $S \rightarrow \mathbb{R}$, defines the "correctness" of each point in the search space. A threshold of this value identifies those points in the space that are considered solutions. This function may be any computable function including a simple boolean function that returns something equivalent to TRUE when a desired object is evaluated. The evaluation function and the n-dimensional search space combine to define an $n+1$-dimensional space, often called a *landscape*, where the value returned by the evaluation function serves to define the height of each point in the search space. Often, descriptions of the operation of search methods are in terms of how they traverse the connectivity of the search space or the "terrain" of the landscape.

2.2 Optimization and Satisficing

When a search is performed to locate global extrema of the evaluation function over the search space with an extremely large number of solutions, such as the space of all possible tours through a collection of cites in the Traveling Salesperson Problem (TSP), the search is typically called an *optimization*. Finding the extrema of such large space is often an NP-Hard problem (Gurari 1989), requiring at least an exponentially increasing number of search space points to be investigated as the problem is scaled up. When the number of objects that must be investigated in order to find the optimal element increases by a factorial, the situation is called *combinatorial explosion*.

Artificial intelligence researchers combat combinatorial explosion in two ways. Often, a tractable subproblem can be defined and solved more efficiently that the original NP-Hard problem. Another way to avoid combinatorial explosion is to settle for less than optimal solutions. In many cases, *near* optimal solutions, solutions within some reasonable percentage of optimal, can be found consistently and quickly. Simon (1969) called this approach *satisficing* to indicate that the solution satisfies minimal requirements and suffices to solve the problem in the environment.

2.3 The Credit Assignment Problem

The problem solving ability of an algorithm is chiefly determined by how "intelligently" the algorithm traverses the possible solutions in the problem space. One possibility is to perform *credit assignment* (Minsky 1963) on the representational components of each solution encountered and bias future solutions towards organizations of representational pieces that perform well together. Credit assignment typically refers to the rating of a structure's components as being beneficial or harmful to solving the problem. The term is most associated with machine learning programs, however, a search algorithm also uses an implicit form of credit assignment when selecting which element in the search space to visit next. Poor search algorithms, such as random search, use inaccurate credit assignment, if any, to guide them through the search space. A more "intelligent" search mechanism embodies an accurate credit assignment mechanism for the task and manipulates only problematic representational components to traverse the space.

Exactly how to assign credit to components of a particular representation is often a difficult problem. Occasionally, if enough assumptions are made, analytical solutions to credit assignment for a particular problem type and representation can be defined, as in the case of neural networks and back-propagation (Rumelhart, Hinton and Williams 1986).

2.4 Weak and Strong Methods

At a very high level, AI problem solving methods can be split into two distinct categories: *weak* and *strong*. A strong method is one that contains a significant amount of task-specific knowledge in order to solve it. In contrast, a weak method requires little or no knowledge about a task. As the name implies, strong methods are more powerful algorithms than weak methods. Strong and weak methods differ in their approach to task applicability and task effectiveness, as shown in Figure 1.

Strong methods can take the form of an analytic solution or an expert system using a large knowledge base. Because their content is directed toward solving a particular problem, they tend to be narrowly applicable, as shown in the figure. However, their task-specific knowledge often allows these techniques to find accurate solutions with minimal computational effort. Analytic methods and rulebases supply the same sort of strength for solving a particular task since they contain the same information at the knowledge-level (Newell 1982). However, general analytic methods and large rulebases are not easily cre-

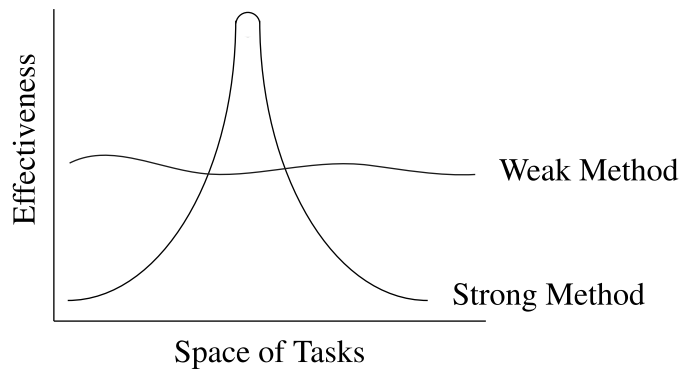

Figure 1: Comparison of the effectiveness of weak and strong methods over the variety of tasks that can be attempted. Strong methods are specialized for a set of tasks and more effective while weak methods are general across tasks but less effective.

ated with current techniques. Knowledge-based AI techniques requires the programmer to extract the task-specific knowledge out of a domain or an expert so it can be inserted into the knowledge base or generalized into an analytical solution.

Weak methods are algorithms that are broadly applicable and extendable to all problem solving instances. AI uses weak methods as general algorithmic shells in which to place knowledge as it is identified and codified. Interactive refinement and/or analytic generalization of the knowledge base is intended to lead to a strong method for the task. In order to be general, weak methods make no commitments to specific operators that manipulate a specific representation (Winston 1983; Rich 1982) and thus make no commitment to credit assignment. At most, they supply a general algorithm for traversing the problem space in an orderly manner. Task-specific knowledge specifies how to traverse the problem space "intelligently" (Winston 1983; Rich 1982). To understand AI's reliance on explicit knowledge it is first important to understand the limitations of weak methods.

2.4.1 Weak Methods

Weak methods are problem solving and search techniques inspired by observations of human performance (Rich 1982; Winston 1983). They include *generate and test*, *hill climbing*, *means-ends analysis*, *constraint satisfaction*, *problem reduction*, *depth-first search*, *breadth-first search* and *best-first search* among others. In this section, I present a

```
function Generate-and-Test;
begin
   obj := Generate();
   while not(Tester(obj)) obj := Generate();
   return obj;
end;
```

Figure 2: Algorithm for Generate and Test weak method.

few representative weak methods with typical properties. In order to simplify the discussion, the operators for the task are assuemd to connect the search space as a tree.

2.4.1.1 Generate and Test

Generate and test (Nerwell and Simon 1972) is the weakest of the weak methods. All other weak methods can be considered specializations of this one method by virtue of its minimal commitments. The algorithm for a generate and test is shown in Figure 2. Two components are needed to define a generate and test search: a generator and a tester. The generator is a function that returns one objects from the search space *S* at each invocation. A generator's properties are typically dependent on the problem and the quality of the operators that organize the set of objects. Winston (1983) suggests that good generators should possess the following properties:

1. *Completeness:* they eventually produce all positions in a search space.

2. *Non-redundancy*: they never damage efficiency by proposing the same solution twice.

3. *Informedness*: they use possibility limiting information to restrict the solutions they propose accordingly.

Random search, the weakest generator possible, selects a single element of *S* with a uniform probability. It therefore does not require an operator set to traverse the space and thus the set of operators, *O,* is null. Note that random search only embodies the completeness principle for good generators but neither of the other two. Because of this minimal ability

```
function Hill-Climber(obj);
begin
   best := null;
   while (not (Tester(obj)) and not(best = obj) do
   begin
      best := obj;
      for opr in O do
         if (evaluate(opr(obj)) >= evaluate(best)) then
            best := opr(obj);
      switch(obj, best);
   end;
   return obj;
end;
```

Figure 3: Algorithm for hill climbing weak method. the switch function switches the values of its arguments so that each has the others value.

to reach everything in the space, random search is often used as a comparison to proposed methods since if a search performs better than random, then some characteristic of the task environment must be present in the generator.

The tester in a generate and test method is very simple. Often it is just a function that takes an object returned by the generator and uses the evaluation function to determine if the object is an acceptable solution. This may involve transforming the object returned by the generator into a more appropriate representation for evaluation. How integrated a generator and tester are determines their efficiency for solving the problem or locating an acceptable solution from the search space. Some of the other weak methods specify their traversal of the search space more explicitly to improve performance while still maintaining a general applicability.

2.4.1.2 Hill Climbing

In hill climbing, the evaluation function plays an important role in the search. Given a current object, this weak method, shown in Figure 3, evaluates each child of the object and selects the one that the evaluation function judges to be best. Thus, it is always the case that the progression of current positions leads to a series of values returned by the evalua-

```
function Depth-First(obj);
begin
   if Tester(obj) then
      return obj;
   else
      for opr in O do Depth-First-Search(opr(obj));
end;
```

Figure 4: Recursive algorithm for depth-first search. O is the set of operators that defines the connectivity of the search space. The connectivity of the space is assumed to be a tree.

tion function that never decreases. Consider the landscape model of the search space. Assuming a maximal value from the evaluation function is preferred, a hill climber traverses the terrain of the landscape so that each step it takes is either up a hill or along a plateau, hence its name. Because the evaluation function determines which child of the current object is the most profitable to investigate next, this weak method has the property of informedness. However, if the evaluation function defines a non-monotonic landscape over the search space, then, depending on the initial position in the space, this weak method halts on some local extrema of the space. Thus, this generator is not complete.

2.4.1.3 Depth First Search

Depth-first search, described in Figure 4, is a stronger form of search that assumes a generator that traverses the search space in a particular manner. The generator begins at an initial position in the search space and expands all of the children of the current structure by calling the algorithm recursively for each child before returning to its parent. Thus the generator moves away from the initial object until the boundary of the search space is seen. Then the algorithm backs up a single operator application and continues. Note that this generator fulfills the completeness and non-redundancy principles of a good generator. However, it doesn't possess a mechanism for integrating information learned in the search to curb future choices. This is because this weak method only uses the tester function to determine if the current structure is a solution. Thus no information is passed between the tester and the generator since the tester is a simple binary function. On average, when searching for a single position in the space, depth-first search visits half of the positions in the search space before identifying the desired object.

```
function Beam-Search(root, beam-length);
begin
   obj-queue := root;
   while not(Tester(head(obj-queue)))
   begin
      for obj in obj-queue do
         for opr in O do push(opr(obj), next-queue);
      sort(next-queue);
      obj-queue := null;
      for i from 1 to beam-length do
         push(pop(next-queue), obj-queue);
   end;
   return head(obj-queue);
end;
```

Figure 5: Algorithm for beam search. *obj-queue* and *next-queue* are standard LIFO queues with standard operators *head*, *push* and *pop*. The sort routine is assumed to use Tester to determine the ordering so that the objects closest to being solutions are at the front of the queue. The space is assumed to be a tree.

2.4.1.4 Breadth-First Search and Beam Search

Breadth first search also traverses the search space in an orderly fashion, as in depth-first search, but does so in a different manner. In this weak method, all of the positions in the space at the same depth away from the initial structure are expanded and evaluated at the same time. If a desired solution is found among them, the search is ended and the object is returned. If not, all objects at the next depth are investigated. Note that this weak method also possesses the properties of completeness and non-redundancy, but as with depth first search, its operation can not be effected by previously rejected.

One disadvantage to breadth-first search is that it can require a significant amount of space. Consider a set of b operators in the set of operators, O. In this space, breadth-first search would need to evaluate b^i positions at depth i, an exponential growth of space. To avoid this, a variant of this weak method, called *beam search* (Lowerre and Reddy 1980), allows at most a fixed number, n, of objects to be queued for subsequent investigation. Figure 5 illustrates the algorithm. It begins by assigning the initial object to the queue of objects and entering the search loop. Inside the loop the algorithm repeatedly expands the last set of objects checked into its children by applying each operator in O. The set of children is then sorted so that better objects are at the front of the queue. Only the best b elements are transferred into the processing queue to be expanded in the next iteration.

For a given $b << |S|$, beam search does not generate all positions of the space. At some point the search must prune some object from the search and thus no child of that node is ever investigated. Thus beam search does not possess the completeness criteria for a good generator. However, it does make use of information during the search to manipulate how it traverses the space. It would appear that beam search trades the completeness of breadth-first search for efficiency and gains the ability to make an informed traversal of the space. In search spaces where the evaluation function is a good indicator of the chance that the desired structure is a child of the current object, beam search is an extremely efficient search method.

2.4.2 Principles of Standard Weak Methods

Weak methods are not intended to be used directly as solutions, but as foundations from which strong methods can be built (Rich 1982). This is because:

> [A weak method's] efficiency when applied to particular problems is often highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. (Rich 1982) (pp 55)

This strong association between weak and strong methods is designed so that as additional knowledge is engineered out of the task it can be used to avoid investigating unpromising objects thus increasing the efficiency of the search (Winston 1983). This is chiefly accomplished either through using the information from the evaluation function more explicitly, the modification of the operator set which defines the connectivity of the space or the addition of *heuristic functions* that estimate the potential of a solution appearing below the current object.

The association of more efficient search through additional knowledge suggests that these methods encode a minimal or non-existent commitment to credit assignment. In breadth-first and depth-first search, credit assignment is ignored. Each of these methods assumes that stronger approaches for deciding what components of an object to alter during the search come from other sources than the basic search space traversal. In hill climbing and beam search, a simple heuristic for credit assignment is used: try all available operators from the current position(s) in the search space and keep the best of the resulting objects. This is best described as a greedy approach to credit assignment.

Another interesting property of weak methods is that they generally apply each operator available to the current object before determining how to proceed. Thus, these methods expand every child associated with a particular node before focusing on a new node of the space. Given this, in order for these methods to be tractable and perform better than random search, the connectivity of the space must be relatively sparse. This observation dictates several properties of the operators that organize the space. First, the number of operators must be small, otherwise the number of children would be a significant portion of the space and the organization provided by the operators would be worthless. In addition, each operator must be one-to-one, i.e. must map from a single point in the space to at most one other point in the space, since an operators is applied only once to an object. This implies that the operators used by weak methods must be deterministic.

These properties of weak methods reflect the fact that these search methods are inspired by observations of human problem solving. In that environment, a human's activities are necessarily sequential ordered and appear to be deterministic, especially when solving problems that are routine. The characterization is that of collecting the possible choices available at the current time, thinking about what to do given this collection of choices and making a decision of how to proceed. Such a model is sufficient for problems where knowledge is readily available and the number of choices from a particular position in the search space is on average small.

2.4.3 Representation of Task Environment in Strong Methods

The task environment is represented in the knowledge that is entered into the weak method to make it strong. If the problem solver's task description is incomplete or the nuances of the task environment eluded the knowledge engineer, then the task description may not be sufficient to solve the problem. This explicit knowledge leads to the problems reviewed in the previous chapter and eventually reduce AI to the search for the "perfect" task environment representation. This representation serves as a model of the task environment and provides an interpretation of the incoming information. If the wrong model of the task is selected, the complexity of the problem in terms of the algorithm can be greater than necessary (Kolen and Pollack 1993, 1994).

Models of the task environment in strong methods often take the form of a knowledge-base within the problem solver. These knowledge-bases encode the task model in any number of representations. The important aspect of the knowledge in the knowledge-base is that it is specific to the task. This means that the motivation for a particular component of the knowledge-base comes solely from the programmer's modeling of the task envrionment. Taken in total, the complete knowledge-base represents a model of the task environment that is internal to the problem solver. Without the motivations of the task environment, the task-specific knowledge that comprises the task model appears as arbitrary associations.

2.5 Evolutionary Algorithms

In general, the weak methods described in the previous sections are generalizations inspired by observations of human problem solving. However, there are many sources of "intelligence" and "search" that can be used to inspire novel search algorithms. One alternative set has come from analogies to physical processes. For instance, Kirkpatrick, Gelatt and Vecchi (1983) use an analogy to physical settling systems for *simulated annealing*. They show that this method can be used to quickly acquire near optimal solutions for difficult, even NP-Hard, problems. Another physical model, called *elastic networks* (Durbin and Willshaw 1985), uses spring equations and a model of gravity to iteratively approach near optimal solutions for similarly difficult problems. Still another method involving a variant of neural networks, called a Hopfield network (Hopfield 1982; Hopfiled and Tank 1985), has been shown to perform in a similar manner. Each of these methods centers on an optimization model that is searching for a minimal "energy" state. Such settling methods work well even when the search space is extremely large and convoluted.

Another possibility is to model the process that gave rise to the intelligent activity of humans as a search and learning method. Analogies to evolution for problem solving, search, function optimization and learning have been proposed under the names genetic algorithms (Holland 1975; Goldberg 1989a), evolutionary programming (Fogel, Owens and Walsh 1966; Fogel 1992), evolution strategies (Rechenberg 1973; Schwefel 1981; Bäck, Hoffmeister and Schwefel 1991) and others. Figure 6 shows the generic algorith-

```
function Evolutionary-Algorithm(population, size);
begin
   for i from 1 to size do
      fitness[i] := evaluate(population[i];
   while not(Tester(bestof(population, fitness)) do
   begin
      Select(population, fitness);
      Reproduce(population);
      for i from 1 to size do
         fitness[i] := evaluate(population[i]);
   end;
   return bestof(population, fitness);
end;
```

Figure 6: Generic algorithm for evolutionary algorithms.

mic shell for an evolutionary algorithm. This algorithm receives as input an array of search space objects, called a *population*, and the number of objects in the array. Like beam search (Rich 1982), this algorithm limits the number of objects that it can retain from one iteration of inspection to another.[1] In general, the initial population is generated to be uniformly distributed in the search space.

On each iteration of the search, each of the objects in the population is evaluated and the best is checked to see if it fulfills the criteria for halting by the Tester function. If not, a subset of the population is selected, based on their fitness, to be the basis of the ensuing population by the Selection function. Copies of these objects replace objects in the population that were not selected for reproduction so that the population size remains constant. Next, the objects in the population are manipulated by applying the reproductive operators. Objects from the previous population are called *parents* while objects created by applying the reproduction operators to the parents are called *offspring*. Each cycle of select-modify-evaluate is called a *generation* of the search.

1. Thanks to Greg Saunders for bringing this to my attention.

The various evolutionary algorithms differ on the model of evolution assumed, how the surving population members are selected, the types of fitness functions used and specific features that are more definitional. The remainder of this chapter explores these distinctions between evolutionary algorithms and trys to sort out the essential differences from those that are non-essential.

2.5.1 Evolutionary Models

The three major variations for evolutionary algorithms differ chiefly in their assumed *evolutionary models*, i.e., what aspects of evolution are important to model to achieve analogous computational effects (Fogel 1994). Genetic algorithms (GA) (Holland 1975; Goldberg 1989a), evolution strategies and evolutionary programming model evolution at different abstract levels. Genetic algorithms emphasize the acquisition of structures. This leads to modeling evolution at the granularity of genetics, i.e. at the level of manipulating the object's structure independent of its interpretation. For evolution strategies (ES) (Rechenberg 1973; Schwefel 1981; Bäch, Hoffmeister and Schwefel 1991), the individual is considered the object of evolutionary optimization. Various behavioral traits of the individual are parameterized and their values evolved as an optimization process. Evolutionary programming (EP) (Fogel, Owens and Walsh 1966; Fogel 1992) uses the coarsest abstraction by emphasizing the adaptation of behavioral traits of various unmixing *species*.

Another distinction between evolutionary algorithms that creates a more pertinent separation is that of search emphasis. Consider Figure 7. When the emphasis of an evolutionary algorithm is on the acquisition of a structure, as in GAs, the reproduction operators manipulate the object so that the structural features of a parent are preserved in the offspring. This allows a broad applicability across many interpretations of the representation but often leads to constructing offspring that have vastly different behaviors from their parents. Likewise, an emphasis on inducing a particular behavior, as in EP, entails using reproduction operators that create offspring that are behaviorally similar to their parent object but may have little structural similarity.

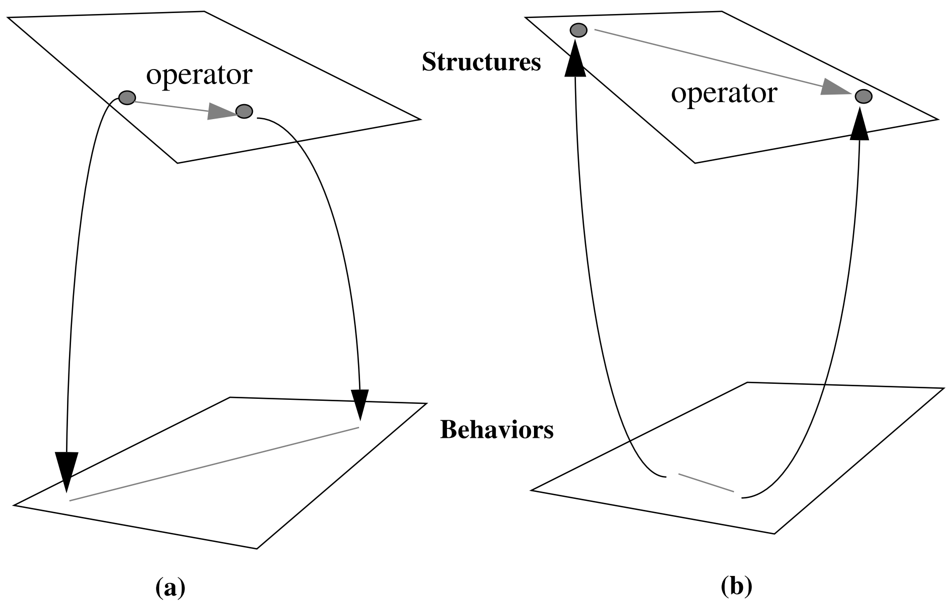

Figure 7: The separation of structure and behavior. (a) Emphesis on acquiring structure results in creating offspring that are structurally similar but may be behaviorally uncomparable. (b) Emphesis on acquiring behaviors might require operators that associate dissimilar structures.

2.5.2 The Selection of Surviving Objects

The selection function for an evolutionary algorithm uses the fitness of the current population to determine which of the objects to use as parents for the next population. While the reproduction function and the evolutionary model are chiefly responsible for the flexibility of an evolutionary algorithm, the selection function largely affects the degree of non-monotonicity in the search by emphasizing either the *exploitation* of good population members or the *exploration* for better ones (Holland 1975; Goldberg 1989a). For searches that promote exploration, the diversity of the population is maintained and the opportunity to locate better solutions increases but at the cost of speed. An emphasis on exploitation naturally decreases the diversity of the population and increases the convergence of the population but typically at a local extrema. Maintaining an appropriate balance between these opposing concepts can be difficult.

Turnover rate signifies what percentage of the population is replaced before evaluating the new offspring. This is referred to as the *generation gap* in genetic algorithms literature (De Jong 1975; Goldberg 1989a). In general, a high percentage of replacement in the population allows more exploration by the evolutionary algorithm while a fewer number of replacements emphasizes exploitation.

Syswerda (1989) describes using a severe turnover rate in his *steady state genetic algorithm* (SSGA) which replaces only a single population members between generations. His results show a marked speed-up in locating solutions. A steady state algorithm has also been shown to speed search in other evolutionary algorithms (Reynolds 1992; Kinnear 1993). Such speed-ups are not surprising given SSGA places an extreme emphasis on exploitation. Consider that this allows a new offspring, if relatively fit, to be recombined more quickly with a variety of other structures, hence quicker exploitation and quicker loss of diversity. Given this emphasis, the quality of solutions for evolutionary algorithms with this feature may be less optimal and more variable than need be for some problems. De Jong (1993) demonstrates these points experimentally.

The determination of the number of times each object is selected to be a parent is either deterministic or probabilistic. Deterministic selection methods have been explored in the evolution strategies literature. The $(\mu+\lambda)$-ES (Schwefel 1981) is an evolution strategy with a population size of $\mu+\lambda$ objects. To begin, the μ objects with the highest fitness are selected with the other λ being discarded. These μ objects are used by the Reproduction function to create another λ objects to bring the population size back to $\mu+\lambda$. Thus, a $(\mu+\lambda)$-ES has a turnover rate of $\lambda/(\mu+\lambda)$. In (μ,λ)-ES (Schwefel 1981), a population of λ objects is maintained. μ of the λ objects are selected and passed to the Reproduction function which creates another λ objects and discards the μ. Therefore, in (μ,λ)-ES selection, no object is in the population for more than one generation giving it a turnover rate of 1.0.

There are numerous techniques for probabilistic selection used in genetic algorithms. *Fitness proportionate reproduction* (Holland 1975), also termed *roulette wheel selection*, calculates the probability of a particular object's selection to be a parent, $\sigma(o_i)$, as:

$$\sigma(o_i) = \frac{f(o_i)}{\sum_j f(o_j)} \qquad \text{(EQ 1)}$$

where $f(o_i)$ is the fitness of object o_i. Each member of the population is selected with a chance proportional to its fitness relative to the total fitness of the population. This allows even the weakest member of the population to occasionally provide offspring for the next generation. Such non-determinism can introduce strong non-monotonicity into the search.

A difficulty with this method is that a large difference between fitness of two objects in the population causes a large difference in selection probability. Such a discrepancy translates into the fittest individual to be selected an inordinate number of times, thus removing diversity. Often when using fitness proportionate reproduction researchers provide a function to scale the fitness of the population members in order to accentuate or de-accentuate the chance objects are selected. Goldberg (1989a) reports that a variety of scaling methods have been investigated in genetic algorithms (Bagley 1967; Rosenberg 1967; Forrest 1985; Gillies 1985). Fitness scaling's benefit is to make the exploitation/exploration trade-off consistent for all potential populations.

Another probabilistic selection function is *tournament selection* which is used by both GAs (Brindel 1981; Wetzel 1983; Koza 1992) and evolutionary programs (Fogel 1992). In a k-competitive tournament, each population member is compared to k other objects, chosen uniformly from the population without replacement. In the GA version, k objects are chosen and the one with the maximal fitness is selected. The EP version compares the fitness of each population member against k other population members, awarding a point for each of the k it is better than. Those objects that score in the top half of the population are then selected and used to create a single object, as in a $(\mu+\mu)$-ES. Competitive methods are preferred since no scaling of the fitness values is required.

While the selection methods described above are generally used within a specific evolutionary algorithm, there is nothing that limits their usage to that particular variation. Experimentation is needed to determine the relative benefits of these various manners of selection across the variety of techniques.

2.5.3 Variations of the Evolutionary Algorithms

The variations of evolutionary algorithms that are of current interest bring differing philosophies of how to algorithmically model evolution. Because of differing commitment to levels of description, each uses a distinct emphasis that leads to a commitment to representations and philosophy of operators. This section describes the assumptions and associated commitments of genetic algorithms, evolution strategies and evolutionary programming.

2.5.3.1 Genetic Algorithms (GA)

The most popular evolutionary model used in current research is the genotypic model preferred by researchers in genetic algorithms (Holland 1975; Goldberg 1989a). Genetic algorithms model a view of evolution as a search for structures, namely for organizations of genes which can be realized into organisms that perform well in a given environment. As a result, the object representation and reproduction operators of genetic algorithms emphasize the manipulation of structure independent of the interpretation of the structure. In addition the results of applying a reproduction heuristic in a genetic algorithm is an object which shares structural similarity to the components is was generated from but which may have significantly different interpretation.

Genetic algorithms are often considered function optimizers (Bethke 1981; Brindle 1981; Goldberg 1989a). However, according to recent investigation and theory, this is erroneous (De Jong 1993; Vose 1993; Fogel 1992). A better characterization of what GAs can accomplish is given by Vose (1993):

> A real GA will with large probability be asymptotically at that local minimum having largest basin of attraction. As the population size grows, the probability of a GA being asymptotically anywhere else converges to zero. (p 64)

Thus a GA only optimizes functions that have the global optima as the largest basin of attraction. Such a property, while not particularly suited to optimization concerns, can be an effective satisficing mechanism.

Representation in a genetic algorithm is organized typically as a fixed-length string regardless of the problem. By definition, this assumes a distinction between the representation that is searched, the fixed-length string, and the representation that is evaluated by the fitness function, which is problem dependent. To map between these two distinct representational spaces, an *interpretation function*, is required. This is often referred to as a *decoding* of one representation into the other (Goldberg 1989a) but in practice the associa-

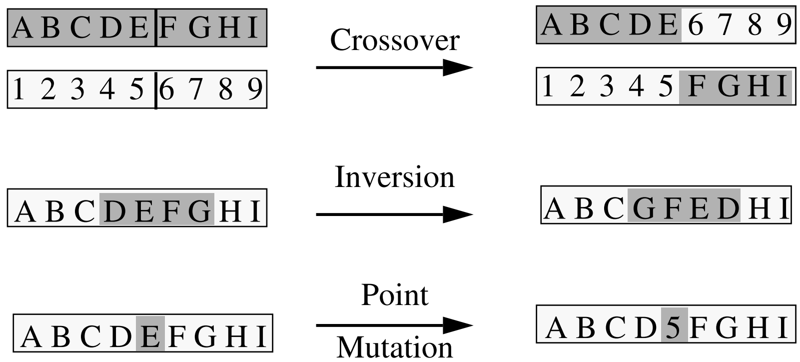

Figure 8: Operators used by genetic algorithms. (a) single point crossover; (b) point mutation; (c) inversion.

tion between the two representations is limited only by computability and creativity of the programmer. In many genetic algorithms, the interpretation function is an indistinguishable component of the fitness function making it hard to identify explicitly.

Often when using a fixed-length structure each individual position of the string can have a specific static interpretation. In such a *positional encoding*, the semantics of a particular representation are signified by its location in the representation. Such a representation can make the evaluation of objects easier but reduces the evolutionary algorithm to an elaborate slot-filling mechanism and artificially limits the size of the search space. Such a limitation is beneficial when it can be guaranteed that the optimal solution is necessarily a member of the reduced search space.

The number of values for a particular position of the fixed-length string is under debate. Holland (1975) suggests that a binary representation is optimal given his interpretation how a GA operates. This interpretation has been disputed (Antonisse 1989; Vignaux and Michalewicz 1993). However, binary strings are generally assumed for theoretical consistency.

The set of reproductive operators used in genetic algorithms are analogous to those thought to promote evolution as modeled by genetics. These operators include *crossover*, *inversion* and *point mutation*, shown in Figure 8. Crossover takes two or more objects of the population and returns objects that are constructed from their contents. This can be a simple, single point exchange, as shown in Figure 8, or an analogues multi-point version (Goldberg 1989a). Syswerda (1989) describes a probabilistic crossover operator that selects components from parent objects with uniform probability. Crossover is the chief manipulation operator for this evolutionary model and is considered the definitional component of a genetic algorithm (Davis 1991).

The other two operators, inversion and point mutation, are applied to the reproducing objects to a far less extent. For inversion, a segment of the fixed-length string is identified probabilisticly, removed from the object and transcribed back into the original positions but in reverse order. This operator is rarely used in practice. Point mutations modify a single position of a string by replacing it with another value at random. In practice, point mutation is performed very infrequently if at all.

2.5.3.2 Evolution Strategies (ES)

The individual model of evolution is typified by evolution strategies (Rechenberg 1973; Schwefel 1981; Bäck, Hoffmeister and Schwefel 1991). In these techniques, the emphasis is not on the acquisition of a structure with a high fitness, as in genetic algorithms, but for a behavior that is rated well by the fitness function. Shifting the emphasis of the search from the structures to the behavior is both beneficial and problematic. By concentrating on optimizing behavior, the representation and reproduction heuristics must create objects that are behaviorally similar to their parents but not necessarily structurally similar. However, if successful, the acquisition of an appropriate behavior should be easier since the effects of interpretation have been modeled in the reproduction operators.

With an emphasis on modeling evolution at the granularity of an individual, evolution strategies consider an individual to be composed of a set of behaviors, each of which is a feature. The interaction between the features is typically unknown. As a result, ES systems use fixed-length, real valued string as representation. Each position marks a separate behavioral trait.

The adherence to fixed-length structures in ES alleviates the problem of how to manipulate the structure in order to preserve behavioral similarity between parent and offspring objects. However, it is not clear how to manipulate the contents of the string to create offspring that are behaviorally similar. This is reflected in the defined operators for ES.

Because ES models evolution at the level of the individual, there is an opportunity for two individuals to be recombined to create offspring. In evolution strategies there are several forms of recombination available. The one most like crossover in genetic algorithms is the *discrete* recombination operator which, given two parents selected from the population, choose the values for the offspring uniformly from both (Bäck, Hoffmeister and Schwefel 1991). This operator is identical to Syswerda's (1989) uniform crossover. Unlike genetic algorithms, ES's real valued strings permit more options for recombination rather than simply remixing the components. The *intermediate* recombination operator creates offspring by averaging the values of the two parent strings (Bäck, Hoffmeister and Schwefel 1991). A third recombination operator, *random intermediate*, probabilisticly determines the evenness of the contribution of each parent for each parameter. (Bäck, Hoffmeister and Schwefel 1991). To address the difficulty of premature convergence in ES systems, each of the recombination operators has a global version that selects a different second parent for each individual parameter of the string while keeping the first parent the same throughout the operation (Bäck, Hoffmeister and Schwefel 1991).

But it is not necessary to manipulate the strings only by recombination. The mutation operator used in ES applies Gaussian noise to each dimension of the representation. The mean is the current value of the parameter. The variance of the noise is determined by the following rule:

> *The 1/5 rule* - The ratio of successful mutations to all mutations should be 1/5. If it is greater than 1/5 then increase σ by a constant, c_i, if it is less than 1/5 then decrease σ by a constant, c_d.

where a successful mutation is one in which the resulting object has a better fitness than its parent. When this rule decreases the variance, the search becomes more focused, with generated offspring being generally closer to their parents in value. When $\sigma$is increased, the search is broadened so that offspring might be further from their parent objects.

Which of the above operators to use on any single act of reproduction is chosen uniformly. In some ES systems, the parameters associated with the algorithm are included in the representation to be modified so their settings can adapt with the strings in the population (Bäck, Hoffmeister and Schwefel 1991). Such approaches have also been investigated in the GA (Davis 1991) and EP (Fogel 1992) liturature as well.

2.5.3.3 Evolutionary Programming (EP)

Evolutionary programming (Fogel, Owens and Walsh 1966; Fogel 1992) employs a a model of evolution at a higher abstraction from both genetic algorithms and evolution strategies. This approach models the reproductive relationship between a species behavior in successive generations (Fogel 1994).

The reproductive operators used in EP are a form of mutation that attempt to preserve behavioral similarity with the parent (Fogel 1992). The motivation for behaviorial simlarity is taken directly from biology where an offspring is generally similar to its parent at the behaviorial level with only slight variations. These variations can be modeled by assuming that the distribution of potential offspring resembles a normal distribution around the parent's behavior (Fogel 1992) in the fitness landscape. Note that a GA's emphesis on structural similarity can not guarantee such a distribution.

Many of the experiments in this dissertation use evolutionary programs as their computational core. Because EP is not committed to a specific representation or operator, the exact nature of the evolutionary algorithm used is described within the description of the experiment.

2.5.3.4 Genetic Programming (GP)

Many of the experiments and explanations to follow in this dissertation involve a variant of genetic algorithms called *genetic programming* (GP). GP is the newest derrivative of genetic algorithms and is currently enjoying an increase in popularity in the community (see Kinnear 1994a). Some researchers have even begun analysing the abilities of GP (Tackett and Carmi 1994) and formulating theories of processing in GP (Altenberg 1994). As defined in Koza (1992), genetic programming has a number of significant properties, both attributed and innate. Some of the attributed properties, especially in comparison to genetic algorithms, are not valid while others have been misdiagnosed or gone unnoticed completely. The following sections provide a description of genetic programming attempts to place the properties of GP into a consistent perspective.

True to genetic programming's name, the individuals of the population are programs. Evolving programs to solve problems seems an obvious application for computational implementations of evolution. However, standard programming languages, like C, Pascal, Fortran etc., are very general, low level and rely heavily on correct syntax to disambiguate the intended behavior. The typical genetic manipulations of genetic algorithms would perform poorly on such languages. Because of the heavy reliance on syntax, straight forward applications of crossover on simple minded representations of programs creates a high prospect for producing programs that are syntactically incorrect. The allows a high probability for creating non-viable offspring which retard the evolutionary progress of the population. Also, the generality of these languages means that even simple task solutions require large programs. Finally, there is no way to anticipate the size of an appropriate solution program.

Koza's genetic programs alleviate these problems by making two basic changes to the standard genetic algorithm. First, each problem solution is a program in a language explicitly designed by Koza for the task at hand. For instance, if the task is to fit a function to a set of data points, Koza designs the programming language, L_1, with an appropriate number of variables and suitable mathematical operators. If, instead, the problem is to create a program to control an artificial ant along a path of food, Koza uses a different language, L_2. in his technique. Essentially, each new problem requires its own distinct programming language. This is problematic since how easy or hard a problem is to solve with Koza's technique is strongly affected by the choice of programming language. Languages that contain primitives that capture significant concepts central to the solving the problem provide overwhelming advantages over languages that are less problem specific. This occasionally brings into question whether the results Koza shows are due to his technique or an artifact of his choice of programming language.

Koza's genetic programs are stored as complex expressions in the provided language with a specific recursive format. Koza uses prefix notation for genetic programs. Given that an expression is the application of an operator to a specific number of operators, prefix notation denotes that the ordering of the expression is the operator followed by the correct number of operands. If in turn each operand is itself an expression, a recursive tree structure develops with the operators as internal nodes of the tree and the operands as the leaves. Borrowing the syntactic convention of LISP, a programming language most associated with artificial intelligence, Koza denotes complete subtrees in his evolved programs by offsetting them with parentheses.

Evolution in genetic programming is carried on the mutability of a specialized crossover operation, first investigated by Cramer (1985), that recombines two parent genetic programs into two syntactically valid children. Recombination simply consists of selecting a complete subtree from each of the parent programs and swapping them to form two offspring. Because all individuals in the population are represented as recursive tree structures, simply removing a complete subtree and replacing it with another automatically preserves all syntactic constraints.

Because expression trees are recursive, they are of variable depth and hence of unlimited size. Although Koza claims his programs take full advantage of this dynamic representation, the code supplied in Appendix C uses a specific variable to limit the ultimate size of the evolving expression trees. Otherwise, the structures can grow without bound and quickly fill the computer's available memory until the time required to execute a single program is prohibitive. Such a limitation is conceptually important when trying to understand the differences of Koza's technique.

2.5.3.5 Comparing Genetic Algorithms and Genetic Programming

Genetic programs are often said to be distinct from simple genetic algorithms due to the dynamic nature of their tree representations and their interpretation as programs. Simple genetic algorithms (Holland 1975; Goldberg 1989a), which use fixed length bit strings, n-point crossover and point mutation are in principle both representationally and operationally general. In fact, there is no theoretical reason why simple genetic algorithms could not construct genetic programs with similar properties. However, as so often happens in science, the practice of genetic algorithms focuses on only a small portion of what is possible.

2.5.3.5.1 Representation of Genotypes

One of the claimed advantages of genetic programs over genetic algorithms is its dynamic tree representation (Koza 1992; Angeline and Pollack 1993a). However, this is not strictly accurate. Consider that a tree with known maximal depth and branching factor is easily represented within a fixed-length bit string. In genetic programs, there is always a practical limitation to tree size in order to keep the problem computationally tractable. Size limitations either come from a defined maximum depth, as in (Koza 1992) and

(Angeline and Pollack 1993a), or from the virtual size of the computer's memory. The representation of the "dynamic" tree is done by simply using only the amount of the fixed-length bit string necessary to represent the tree. As the tree grows, more of the bit string is used. Jefferson's group (Jefferson et al. 1992) use such an approach in their genetic algorithm that evolves finite state machines. The designation of the expression tree representation as being dynamic is then only a logical convenience and offers no actual representational advantage. However, dynamic allocation of computational resources on an as needed basis does allow an implementational advantage over extremely large fixed-length strings.

2.5.3.5.2 Complexity of Interpretation

When using in genetic algorithms, the fitness function uses an *interpretation function* to convert the bit string representation into the evaluated form. For instance, in the classic experiments of De Jong (1975), the bits were interpreted as binary numbers and passed as parameters to the objective functions being optimized. What makes genetic algorithms general is that they manipulate the bit strings without any regard to their eventual interpretation.

Interpretation functions are typically very simple, usually a conversion from a bit string to a real number as in (De Jong 1975). The typical interpretation function uses a *positional encoding* (Angeline and Pollack 1993a; Angeline and Pollack 1992) where the semantics of the interpretation are tied to the position of the bit in the genotype rather than its content. As a result, the interpretation of a bit string is likely to be a simple combination of the various positions, similar to a union of independent features. Positional encodings also force the programmer to organize the representation so that bit positions with related semantics are reasonably adjacent and have a lower probability of being separated by a crossover operation. In so called "messy" genetic algorithms (Goldberg, Korb and Deb 1989) where the organization of the genotype is allowed to evolve, a marker designates the "original" position of the allele and thus its semantics.

In genetic programming, the interpretation of a particular expression tree is performed without consideration to the position of the allele. Thus, as in computer programs, the interpretation of the allele "if" in a genetic program is the same regardless of its position in the expression tree. The arbitrarily complex association between an expression and its interpreted behavior allows for a unlimited variety of dynamics to emerge naturally from the evolutionary process. Again, while there is in principle no reason why a standard genetic algorithm could not use a similar interpretation functions, this has not been the practice.

2.5.3.5.3 Syntax Preserving Crossover

While it is always true that manipulating a bit string representation with operators that disregard its interpretation is the most general method of evolving solutions, to assume it is invariably the most accurate and the most efficient method is counter to both intuition and experimental results. Specialized genetic algorithms such as (Grefensttete 1987), (Michalewicz 1993) and other evolutionary computations such as evolutionary programs (Fogel 1992; Angeline, Saunders and Pollack 1994) make great use of operators that are more friendly to the representation's eventual interpretation. In essence, these evolutionary computations manipulate the evaluated representation more directly. This approach either eliminates the interpretation function or compensates for it with appropriate operators.

Genetic programming uses a similar strategy by employing a crossover operator that is sensitive to the expression tree structure it is manipulating. This simple crossover scheme preserves the syntactic constraints of the representation by swapping only complete subtrees between parents. Adding the ability to identify complete subtrees to GP's crossover operator ensures that only syntactically valid programs are constructed, removing all syntactically invalid programs from the search.

In actuality, GP's use of a crossover operator that preserves the syntactic constraints of the representation language is the only true distinction between standard genetic algorithms and genetic programming. More importantly, the conceptual conveniences of a dynamic representation, non-positional interpretations, and direct manipulation of representation, highlights the narrow focus of genetics algorithms research. Genetic programming's most significant contribution to the field may be to re-emphasize the variety of options not yet explored.

2.6 When is a Weak Method Not Weak?

Evolutionary algorithms are currently popular engineering tools and are being applied to a wide range of problems. Their appeal is an apparent applicability to problems where little knowledge is available earning them a reference as “strong weak methods.” The next chapter explores what about evolutionary algorithms is unique from weak methods and how this improves their ability to search complex spaces more effectively. This leads to the identification of a new method for credit assignment that allows task-specific knowledge to emerge naturally from problem solving process and provides the basis for emergent intelligence.