Searching for Search Algorithms: Experiments in Meta-search

Brian J. Ross

Technical Report # CS-02-23
December 2002

Submitted for publication.

Searching for Search Algorithms: Experiments in Meta-search

Brian J. Ross

Brock University, Department of Computer Science, St. Catharines, ON, Canada L2S 3A1

Abstract

The conventional approach to solving optimization and search problems is to apply a variety of search algorithms to the problem at hand, in order to discover a technique that is well-adapted to the search space being explored. This paper investigates an alternative approach, in which search algorithms are automatically synthesized for particular optimization problem instances. A language composed of potentially useful basic search primitives is derived. This search language is then used with genetic programming to derive search algorithms. The genetic programming system evaluates the fitness of each search algorithm by applying it to a binary-encoded optimization problem (Traveling Salesman), and measuring the relative performance of that algorithm in finding a solution to the problem. It is shown that the evolved search algorithms often display consistent characteristics with respect to the corresponding problem instance to which they are applied. For example, some problem instances are best suited to hill-climbing, while others are better adapted to conventional genetic algorithms. As is to be expected, the search algorithm derived is strongly dependent the scale and representation of the problem explored, the amount of computational effort allotted to the overall search, and the search primitives available for the algorithm. Additionally, some insights are gained into issues of genetic algorithm search. A novel "memetic crossover" operator was evolved during the course of this research.

Key words: Meta-search, Heuristics, Genetic Algorithms, Genetic Programming
PACS:

Email address: `bross@cosc.brocku.ca` (Brian J. Ross).
URL: `http://www.cosc.brocku.ca/ bross/` (Brian J. Ross).

1 Introduction

One important and practical contribution of artificial intelligence research to computer science is the notion of search[11]. Any introductory AI textbook will devote a number of chapters to search techniques, from general search notions such as blind and heuristic search, to more advanced paradigms such as minimax, evolutionary algorithms, tabu, simulated annealing, and neural network training [19][14]. The challenge in using search to solve a problem is to find the most effective strategy for a particular problem at hand. Different search algorithms tend to show different performance characteristics for different problems. The selection of an appropriate algorithm, and fine-tuning its parameters, often depends on the practisioner's experience with the problem being analyzed.

This paper proposes a novel approach for finding an effective search strategy for a given problem instance. Using genetic programming, we evolve search algorithms for specific instances of optimization problems. This is an instance of meta–evolution: (i) at the top meta–level, we are evolving search algorithms; (ii) at level two, each search algorithm is applied to a problem instance, to search for a solution for that problem. The meta-level search algorithm language is composed of various primitives useful in a search algorithm context. Primitives include operators for selection, alteration, and replacement of members of the search set. A candidate search algorithm is executed upon the level 2 optimization problem. The search algorithm is repeatedly applied to the problem until a solution of acceptable quality is found, or a maximum number of iterations is reached. The genetic programming procedure evaluates the fitness of a search algorithm by inspecting the best solution in the set of level 2 candidate solutions.

There are a number of motivations for this research. Firstly, the plethora of search algorithms found in the literature are all instances of computer programs. A practical way to describe a search technique, and contrast it to another, is via code in a programming language. For example, a hill-climber, beam search, random search, and genetic algorithm are definable by appropriate programming code. It is interesting to consider whether such search algorithms might be automatically synthesized. More importantly, can *effective* algorithms be synthesized for particular instances of search problems? Hence, this level of meta-search differs from most work in meta-heuristics, in that an entire algorithm is sought, rather than a parameter vector of search heuristics.

Another motivation of this work is to gain some insight into the interrelationships of the methods used within search algorithms, and how they relate to the problems to which they are applied. The success of a search strategies for solving a problem is strongly dependent upon the ability of the search to exploit regularities of the problem at hand. The search algorithms evolved in this paper can usually be rationalized by examining how their primitive structures correspond to the characteristics of the problem they are being applied upon.

Some background issues regarding search and meta-search are discussed in Section 2. Section 3 describes the experimental design used in this research. A variation of the Travelling Salesman problem is described, and the genetic programming system is outlined in detail. A summary of the main experiments undertaken is given. The results of the experiments are reported in Section 4, and further analysis is given in Section 5. Section 6 summarizes the paper.

2 Background

Search is an intrinsic means by which computers can find solutions to problems. For problems with small search spaces, search can be an effective way to obtain a solution, especially when a known search algorithm can be used. This means that one can avoid writing a new algorithm specially designed for the problem of concern. Search is also a useful means for finding solutions for problems that are lacking efficient algorithmic solutions. Many NP-complete problems fall into this category. Although search does not solve intractable, complex problems, in many cases it will find “acceptable quality solutions” – and often much better than what is possible with enumeration.

A conventional approach to solving a given search or optimization problem is to apply a suite of search algorithms and parameterizations, and perform a comparative empirical analysis of the results [18]. Intuitively, if a particular search algorithm shows considerably better results, that search algorithm can be said to be more naturally adapted to the fitness landscape defined by the search space of the problem – at least in comparison to the alternative search techniques tested.

The need for exploring a variety of search techniques for a problem stems from the basic fact that, for most problems of interest, search is difficult. The No Free Lunch (NFL) theorem by Wolpert and Macready states that, over the space of all search problems, no search algorithm is superior over random search or enumeration[20]. Similar results have been shown for optimization problems [21]. Although the NFL theorem is theoretically irrefutable, a counter-argument to its implications is that the space of all search problems is of little interest. Rather, real-world problems of concern to most people have *structure* and *regularities*, and define explorable, if not challenging, search spaces.

In restating the NFL theorem for evolutionary computation, Culberson also points out that the implications of computational complexity theory are even stronger than the NFL theorem [2]. NP-complete and other intractable problems have structures that are totally well-defined. This structure cannot be exploited by search – or any known algorithm – in solving NP-complete problems. Nevertheless, intractable problems (such as the famous Travelling Salesman Problem used in this paper) are often applied as test problems for search algorithms and other soft computing techniques. Although a solution to such problems will not be forthcoming in most instances, we are often interested in obtaining *reasonable* solutions to known cases. This is a motivation for work in the soft computing field, in which heuristic methods are used for finding approximate solutions to difficult problems. Thus, search algorithms can be extremely practical for finding good answers to difficult (intractable) problems. The challenge, however, is finding a good search algorithm for the problem at hand.

The use of meta-heuristics for search is an attempt to automate the discovery for useful search algorithm paradigms and parameterizations. Within genetic algorithms, one approach is to include search parameters within the search problem representation, and let the search mechanism find a useful parameter configuration for a problem. This is called *self-adaptation* in genetic algorithm work. A typical example is the work of Fogarty [4], in which mutation rates adapt during the run. Liang, Yao and Newton discuss the self-adaptation of step sizes as used in evoltionary algorithms [10]. Ombuki, Nakamura and Onaga in applying a genetic algorithm towards the NP-complete Job Shop Scheduling problem [13]. In their gkGA algorithm, three different heuristic strategies for resolving deadlocks in scheduling solutions are encoded in the chromosome. These strategies are subject to the same evolution effects as the rest of the problem representation. They show that better overall performance is obtained by permitting the the search algorithm to discover the appropriate deadlock heuristic.

Meta-search or meta-evolution has also been studied in the context of genetic programming, and usually in regards to evolving new variation operators. Teller uses co-evolution to generate reproduction operators to be used by a main program population evolving in parallel [17]. Angeline [1], and Iba and de Garis [7], use self-adaptation to evolve crossover operators that adapt to programs during the run. Edmonds investigates the co-evolution of variation operators such as crossover [3]. In all these papers, meta-evolution of variation operators that adapts to the particular problem at hand is conducive to better performance, when compared to the use of generic operators.

Perhaps the most ambitious example yet of meta-search is the work by Spector and Robinson [15]. They apply genetic programming to a stack-based language, in which programs have the ability to access and process themselves.

One goal of that research is to investigate whether evolutionary behaviour itself can be evolved.

3 Experiment

The remainder of the paper uses the following terminology to describe the search algorithms that are being evolved by the genetic programming system. We wish to use these generic terms to describe search algorithms, in order to remove any apparent terminological bias when describing the algorithms evolved by genetic programming.

- *search space*: The universe of solutions to some problem of interest.
- *individual*: A single point in the search space that is currently known by the search algorithm. Synonyms include state, point, candidate solution, or hypothesis.
- *representation*: This is the denotation of the search space for a particular problem of interest. Individuals are encoded using the representation for the problem. The representation is called a chromosome or genotype in genetic algorithms.
- *search set*: This is the finite set of individuals currently under investigation by a search algorithm. This search set is called the population in genetic algorithms.
- *variation operator*: This is an operator which, when given an individual encoded with some representation, creates a variation of that individual. Synonyms include next-state operators and reproduction operators.
- *score*: This is a measurement of the relative merit or quality of an individual. A score is obtained by applying a problem domain-specific objective function or evaluation function to an individual. A score is called the fitness in evolutionary computation.

For convenience, we will borrow a few genetic algorithm terms, such as “parents” and “offspring”, when discussing crossover variation operators.

3.1 Travelling Salesman Problem

The NP-complete Travelling Salesman Problem (TSP) is used as a test problem for all the experiments in this paper [5,6]. It can be informally described as follows: *Given a graph of nodes and edges, what is the minimum distance of the path that starts at an initial node, visits each node once, and returns to the initial node?* The TSP variant used here uses the following conventions: (i) the nodes lay on either a 4-by-4 or 8-by-8 grid; (ii) the nodes are maximally

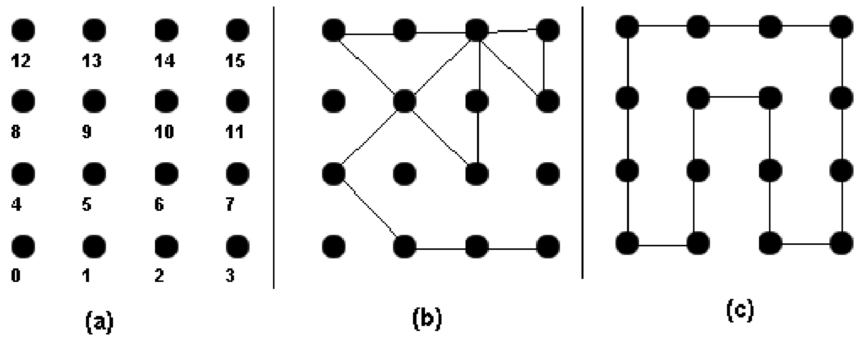

Fig. 1. 4x4 TSP: (a) grid; (b) erroneous path; (c) one solution path.

connected (ie. there is an edge between each node and every other node in the graph); and (iii) the distance between horizontally and vertically adjacent nodes is 1. Figure 1 shows a 4-by-4 grid, a non-optimal and erroneous path, and a solution path (length 16). Each city in the grid is given a numeric label, from 0 to 15 (or 63 for the 8x8 grid).

The TSP has been a traditional "lab rat" for heuristic search experiments, as it is an excellent problem for studying the relationship between problem representation and search performance [16]. We use two well-known binary representations for the TSP. The representations for the 4-by-4 grid are as follows; the 8-by-8 representations are similar.

- *Direct*: The representation uses a 64-bit string. Each contiguous 4-bit field denotes a city label, from 0 to 15. A path is decoded by reading the city label fields left-to-right along the bit string.
- *Indexed*: The representation uses a 64-bit string. Each contiguous 4-bit field denotes an index or jump interval. A path is decoded as follows. A list of unvisited cities is first composed. A list pointer into the list is set to the first city in the list. The first jump index K_1 is read from the left of the string. Starting at the list pointer, the K_1 city down from the list pointer is found (the city at the top of the list is at K=0), added to the path, and removed from the unvisited list. The following K_2 index is read, the K_2 city from the current list pointer is added to the path and removed from the list. If at any time the list pointer goes beyond the end of the list, it is set to the start. This continues until all the cities have been visited.

With respect to genetic algorithms, the indexed representation performs much better than the direct one. One reason is because the direct representation, albeit simple, does not guarantee the denotation of a legal TSP path. The list of city labels within the representation can contain duplicate cities, which are illegal in the definition of the TSP. For every duplicate city, there will

I ::= *Select* | *Replace* | *Growth* | *Variation*

Select ::= sel_rand | sel_best | sel_worst | sel_tourn_best(*N*)
| sel_tourn_worst(*N*)

Replace ::= rep_rand(*I*) | rep_best(*I*) | rep_worst(*I*) | rep_tourn_best(*N*,*I*)
| rep_tourn_worst(*N*, *I*)

Growth ::= add_rand(*I*) | add_best(*I*) | add_worst(*I*) | add_tourn_best(*N*,*I*)
| add_tourn_worst(*N*, *I*)

Variation ::= crossover(*I*,*I*) | crossover2(*I*,*I*) | crossoverN(*I*,*I*) | mutate(*N*,*I*)
| mutateX(*N*, *I*)

N ::= score(*I*) | iter_num | pop_size | *N Op N* | ERC

Op ::= min | max | + | - | / | *

Fig. 2. Search Language Grammar

be a corresponding city missing from the path. Illegal paths can arise during random population generation, and during reproduction with crossover and mutation. One solution is to repair the path after random generation or reproduction. We choose to leave illegal paths alone, and penalize them during their evaluation.

The indirect representation always denotes legal paths. It is also the case that the genetic algorithm crossover operator works better with the indexed representation, due to the fact that it better preserves and transfers information content between parents to offspring.

3.2 The Search Algorithm Language

A simple search language is now introduced. This language is intended to be one with which a variety of basic search algorithms might be derived. It is in no sense a general language, as the only possible useful algorithms derivable in it are search strategies. It is also not attempting to be comprehensive over all possible search algorithms.

Before discussing the specifics of the language, a few assumptions about the search environment must be clarified. In our experiments, individuals are binary strings denoting TSP paths, as described in Section 3.1. Search programs use a search set of individuals, which is a finite set of cardinality ≥ 1 from which the search will explore. It is assumed a search set has known minimum and maximum size bounds. When a search program is executed, it will perform a single alteration to the search set – either replace an individual in it, or add an individual to it. Alternatively, a search program might merely return an individual expression with no alteration to the search set. In this case, we will either: (i) if the set size is less than the maximum bound, add the individual to the set; or (ii) if the set size has reached its maximum, replace a random individual in the set with the expression.

Table 2 shows the full grammar of the search language. We will use subsets of this grammar during experiments. In the grammar, terms without arguments are terminals, and the others are functions. Expressions are evaluated with eager, left–to-right execution of arguments.

Select operators are those used to select an individual from the current search set of individuals. The select operators include selecting a random individual, the strongest individual, and the weakest individual, as based on each individuals current score or distance to a solution. Selection also includes two tournament selection operators. *Sel_tourn_best* performs selection by randomly selecting N individuals from the search set, and keeping the one with the best score. The size is calculated from the floating point argument by the following expression:

$$tournsize = (mantissa(|Nval|)\ modulo\ 6) + 2$$

wheren Nval is the computed value of the numeric argument expression. This returns an integer between 2 and 7. *Sel_tourn_worst* is similar, except that the weakest individual in the tournament is selected.

Replacement operators take an individual expression, and replace an existing search set member with it. Growth operators first try to add their individual expression argument to the current selection set, so long as the set is smaller than the maximum bound. If the set has reached its maximum size, the growth operators revert to replacement operators. The semantics of how individuals are selected to be replaced are the same as the select operators. However, only one search set alteration (replacement or growth) is permitted during a single iterated execution of the program. The last replacement or growth operator executed during the normal execution order of the program is designated as the single alteration operation for the program, and earlier replacement or growth operations are ignored. In addition, the actual expression that is added or replaces a population member is the final expression discovered by the whole program during that iteration. Hence, in the following program,

```
(crossover
    (rep_worst sel_best)
    (rep_tourn_best sel_best))
```

the *rep_tourn_best* operator is the last one interpreted, and so it will be the replacement operator for the expression. The individual it will replace into the search set is the result from the top-level crossover expression.

Variation operators are the basic operators for exploring the search space. They are essentially "next state" operators that create a new state from a given position in the search space. We borrow two types of variation operators from genetic algorithm technology. The mutate operators create random variations of an individual expression. The *mutate* operator flips each bit of an argument expression with a probability P. For example, if the probability is 5%, then there is a 5% chance that each bit is flipped. The probability is computed from the argument N by computing the absolute value of its fraction component:

$$|mantissa(Nval)|$$

MutateX is similar, except that each bit has a probability P of getting a random value. In other words, the probability of bit mutation is half that of *mutate*.

The crossover operators create a single new individual using two other individuals ("parents"). The *crossover* operator performs single-point crossover: finds a random position in a path representation, and swap each portion around the crossover point between the parents to create offspring. Note that only one of the two potential offspring is retained and returned as a result. *Crossover2* performs 2-point crossover, in which two random crossover points are used to swap individual substrings. *CrossoverN* performs N-point crossover. A random bit mask of the length of the path representation is generated, and the mask bits determine which bits are swapped or not.

The remaining portion of the language is used for creating numeric expressions. The *score* operator returns the objective score of its argument expression. *Iter_num* is the current iteration number of the search execution, and *set_size* is the size of the current search set. Some standard mathematical and comparison operators are included. Finally, *ERC* terminals are ephemeral random constants [8]. These are randomly initialized numeric constants that retain their values throughout the lifetime of the genetic programming run.

Some example programs are given in Figure 3. The first program shows one possible random search strategy: select a random individual, flip half its bits, and replace a random individual with it. The second program shows one style of hill-climbing: select the best individual in the search set, randomly flip

Random search

```
    rep_rand(mutate(0.5, sel_rand))
```

Hill-climbing

```
    rep_worst(rep_best(mutate(0.5, sel_best)))
```

Annealing

```
    rep_worst(rep_best(mutate(0.8/iter_num, sel_best)))
```

Genetic algorithm

```
    rep_tourn_worst(3,
        mutate(0.05,
            crossover(sel_tourn_best(4), sel_tourn_best(4))))
```

No search

```
    rep_best(sel_worst)
```

Fig. 3. Example Search Programs

its bits, and replace the worst individual in the search set with the result. Note that only the first replacement operator (*rep_best*) is used, as the other is executed earlier, and is hence ignored. The annealing program shows how mutation is reduced as the iteration steps increase. The genetic algorithm is a generic one. An offspring is created via crossover with two parents selected via score-proportional selection. The result is given a slight mutation, and the result replaces a weak individual in the search set. The final example shows a nonsensical algorithm that is possible in the search language. This search stagnates on the weakest individual in the search space currently visited.

3.3 Evaluation of Search Algorithms

A candidate search program is evaluated as follows. A random initial search space for a given problem of interest is randomly generated. This is performed once per genetic programming generation, and therefore all search algorithms will be evaluated on the same initial search set. The program is iterated on the problem set up to a maximum iteration limit. The score of the best TSP path in the search set is used as the fitness of the search algorithm. Note that the alternate strategy of using the mean score of the entire search set is unwise, because search programs will try to repeatedly clone the fittest TSP paths in the search set, in an attempt to boost their overall fitness.

The score of a TSP path depends on whether the direct or indexed representation is used. The direct representation computes the length of the given path in the bit string, and subtracts from it the minimum path for that grid (16 or 64, respectively). It then adds a penalty for missing cities in the path, adding 5 (for 4x4 grids) or 12 (for 8x8 grids) per missing city. For the indexed representation, the path length minus the minimum path length is used as the score.

Preliminary experiments showed that a single search algorithm's performance often varies dramatically in different search runs. The nature of heuristic search means that different initial locations in the search space may produce different results, entirely due to random chance. For a fixed problem instance with randomly initialized data sets X and Y, program A run on X may produce different results than when run on Y. Additionally, algorithm A can produce varying results when executed on a single data set X in different runs, due to the stochastic nature of operators used in the algorithm, such as *mutate* and *sel_tourn_best*. The most accurate evaluations of search algorithm performance requires statistically significant sized test sets. Unfortunately, for even the simpler TSP problems we investigate, this is impractical within a genetic programming context. A genetic programming run requires the evaluation of thousands of programs, and prolonged evaluation of programs makes genetic programming search too lengthy to perform.

To help address this problem, we perform a more thorough analysis on the best search algorithm in each genetic programming generation. The best algorithm of each genetic programming generation is tested on five additional random TSP search sets, and the mean score for the five resulting scores and the original program score is calculated. The best solution for a genetic programming run is designated as the search algorithm that received the best overall average score during the entire run.

3.4 Genetic Programming Details

The lilGP 1.1 system is used in the experiments[22]. This is a C-based genetic programming system that implements tree-based genetic programming [8]. Strong typing is used for defining the individual and numeric types in the search language in Figure 2 [12].

GP Parameter	*Value*
Evolution paradigm	generational
Max generations	100
Population size	1000
Runs/experiment	10
Initialization	ramped half&half
Initial tree depth	2 to 6
Max tree depth	17
Crossover rate	0.95
Mutation rate	0.05
Tournament size, crossover	4
Tournament size, mutation	7
Probability internal mutation	0.10
Probability external mutation	0.90

Table 1
Common genetic program parameters

Table 1 shows the genetic programming parameters used in all experiments. Most of these parameters are standard in the literature[8]. The initial genetic programming population is generated using ramped half&half tree generation. Here, half the trees generated are *grow trees*, in which a terminal or nonterminal can be randomly selected as the root of each subtree, while the remaining half are *full trees*, in which nonterminals are always selected so long as the tree depth limit is not exceeded. During tree generation, the tree depths are staggered (or *ramped*) from depths 2 through 6. The result is a population of random expressions having a varied distribution of tree shapes. The probability of applying crossover is 95%. When applied, parents are selected using a tournament of size 4. When mutation is used, there is a 90% probability that tree leaves will be mutated.

Table 2 shows some of the search problem variations investigated. Three general sets of experiments are undertaken. Experiment set A restrict the variation operators to single-point crossover and mutation. No growth operators are used, and the search set is kept at a constant size of 100. Experiment set B also restricts the growth operators, and keeps the search set size to 100. However, the full set of variation operators are possible. Experiment B also tries various maximum iteration limits on different TSP sizes. Experiment set C focusses on the effect of growth operators. These experiments begin with an initial search set of size one.

Common parameters

Replacement:	all
TSP representations:	direct, indexed
TSP sizes:	4x4, 8x8

A: Restricted variation, no growth

Variation ops:	crossover, mutate only
Growth:	none
Terminals:	all except *set_size*
Iteration limit:	1000
Search set size:	100

B: Unrestricted variation, no growth

Variation:	all
Growth:	none
Terminals:	all except *set_size*
Iteration limits:	250, 1000, 3000
Search set size:	100

C: Unrestricted variation, growth

Variation:	all
Growth:	all
Terminals:	all
Iteration limit:	1000
Search set size:	min 1, max 100

Table 2
Search experiments

4 Results

The primary interest in our experiments is to analyze the characteristics of evolved search algorithms for different TSP variations. In order to perform an analysis of the search algorithms, some post-processing of the results had to

Table 3
Summary of algorithms

	Experiment	Best soln fitness	Hill-climbing	Annealing	GA	Memetic crossover
A	direct 4x4	8.27	5	2	-	3
	direct 8x8	177.13	6	4	-	-
	indexed 4x4	2.65	-	-	-	10
	indexed 8x8	114.68	-	-	-	10
B	direct 4x4 250	9.88	4	-	-	6
	direct 4x4 1000	8.25	-	-	1	9
	direct 4x4 3000	5.26	-	-	-	10
	direct 8x8 1000	172.22	2	4	3	1
	indexed 4x4 250	4.72	-	-	1	9
	indexed 4x4 1000	2.72	-	-	-	10
	indexed 4x4 3000	1.39	-	-	-	10
	indexed 8x8 1000	101.75	-	-	-	10
C	direct 4x4	8.51	4	-	6	-
	direct 8x8	178.48	1	8	1	-
	indexed 4x4	3.46	-	-	4	6
	indexed 8x8	122.67	-	-	2	8

be undertaken:

(1) The best individual in each run was identified via the average score over 6 separate TSP data sets (see Section 3.3).
(2) The solutions from step 1 were simplified, by hand-editing them to remove dead code (program bloat).
(3) Frequency counts of program operators were then obtained from the simplified programs.

An overview of the results is in Table 3. The summary identifies 4 broad categories of search algorithms identified in the results (10 runs per experiment). The following broad categories are used to classify search algorithms:

- *Hill-climbing*: mutation operators are exclusively used, and they are used expressly on the best individual in the search set (*sel_best*).
- *Annealing*: similar to hill-climbing, except that the mutation rate diminishes as the search proceeds. The rate is usually inversely proportion to the itera-

tion step or search set size (in terms of the growth experiments in C). This is similar to the reduction of mutation rates used in simulated annealing.
- *GA*: no more than 2 crossover operators are used, possibly in combination with mutation. With the single search set used in our searches, this would be a *steady-state genetic algorithm*.
- *Memetic crossover*: 3 or more crossover operators are used, possibly in combination with mutation. The use of multiple crossovers from the search set is akin to a memetic search [9], in which a number of individuals in the search set contribute information to the next individual created. Repeated introduction of new information by multiple individuals via crossover has the statistical effect of introducing many instances of useful new information, while simultaneously correcting erroneous or poor information obtained by some.

We use the above classification labels fairly loosely; for example, annealing alludes to the reduction of mutation rate found in simulated annealing searches. Note that there are possibly other specialized variants of these general algorithm categories, but they were not analyzed nor tabulated.

Some general trends in Table 3 are clearly seen. As expected, the scores for the 4x4 TSP grids are lower than the more difficult 8x8 grids. The indexed representations outperform the direct representation. In experiment set B, permitting the search algorithms higher iteration limits results in better performance.

Definite patterns are evident in the distribution of evolved algorithms. In all the experiments, the indexed representations are biased towards crossover-based search strategies (GA, memetic), while direct representations favour mutation-based search strategies (hill-climbing, annealing). This is due to the fact that the indexed representations retain structure, whereas the direct representation is prone to noise and error. This implies that mutation will be more productive on the direct representation, and crossover is more productive on indexed paths. The non-growth experiments (A, B) tended to avoid GA-style search, with one exception being with direct 8x8. The growth experiments (C), however, favoured GA-style search in 3 of the 4 experiments.

It is interesting to note that whenever mutation is used as the exclusive variation operator (hill-climbing or annealing), it always uses the selection of the best individual in the search set. No mutation-based algorithms were evolved that used tournament or random selection of the next search state to explore. On the other hand, it was found that programs that favoured the use of crossover also tended to use tournament selection.

Whenever mutation was used in experiment A and B, the rates were usually very low (less than 5%). An exception was that very high rates (95This had

Table 4
Variation operator distribution in experiment B runs

		Direct		Indexed	
		Incidence	Program	Incidence	Program
Experiment	Operator	avg	freq	avg	freq
4x4 250	mut	0.56	0.5	0.15	0.5
	mut/i	-	-	-	-
	mutX	0.49	0.6	0.23	0.5
	mutX/i	0.13	0.1	-	-
	cross	0.13	0.5	0.38	1.0
	cross2	0.25	0.5	0.30	1.0
	crossN	0.37	0.6	0.20	0.8
4x4 3000	mut	0.09	0.4	0.15	0.7
	mut/i	-	-	0.14	0.1
	mutX	0.17	0.9	0.11	0.6
	mutX/i	-	-	-	-
	cross	0.19	0.9	0.35	1.0
	cross2	0.41	0.9	0.34	1.0
	crossN	0.30	0.9	0.21	0.6
8x8 1000	mut	0.64	0.2	-	-
	mut/i	0.69	0.5	0.05	0.1
	mutX	0.51	0.7	-	-
	mutX/i	-	-	0.06	0.1
	cross	0.14	0.1	0.49	1.0
	cross2	0.33	0.1	0.45	1.0
	crossN	0.40	0.3	0.11	0.5

the effect of flipping the path within the representation, and thus preserving inherent information. The experiment C runs usually required more significant mutation rates, which is necessary in order to grow the search set from the initial set of a single individual.

Some further analysis of the distribution of variation operators found in some experiment B runs is given in Table 4. The operators listed include all the variation operators. The *mut/i* and *mutX/i* are annealing mutation operations:

```
(add_tourn_worst 2
      (crossover2
            (crossover
                  (crossover
                (sel_tourn_best 2)
                        (mutate 0.48448
                              (mutateX (/ (+ set_size 0.55210)
                                          (- set_size 0.06402))
                                 (crossover2
                        (crossoverN sel_best sel_worst)
                        sel_rand))))
                  (sel_tourn_best 2))
            (mutateX (/ 0.33870 iter_num) sel_best)))
```

Fig. 4. Example annealing search solution: experiment C (growth), 8x8 indexed

those whose mutation rates decrease as the search proceeds. Incidence average is a measure that answers the question: if an operator is found in a program, how often is it used? It can be thought of as an "operator density" measure, which is averaged over all 10 solutions per experiment. The program frequency column refers to the proportion of solutions within which the operator can be found. Multiplying the incidence average and program frequency values together gives the total frequency of the operator within the entire set of runs for that experiment.

Looking at this table, one result is that in all the indexed representations, N-point crossover is used less frequently than 1-point and 2-point crossover. This is likely due to the fact that the N-point crossover is overly destructive of information structures encoded in the indexed path representations. The opposite holds for the direct representation. With this noisier, more error-prone representation of paths, N-point crossover behaves more like a mutation operator. This is especially evident in the direct 8x8 case, whose runs strongly favour mutation. Another observation is that, when mutation is used in the indexed runs, it plays a minor role compared to the crossover operators, and is negligible in the 8x8 indexed runs. Finally crossover becomes more prevalent in the 4x4 3000-iteration direct runs than the 4x4 250-iteration direct. This suggests that crossover variation has more opportunity to explore useful patterns when more iterations are permitted in the search.

The growth runs in C were performed in order to see what general styles of search might evolve for search sets consisting of a single random individual. It was found that annealing-style mutation was negligible in the 4x4 grid, but more prevalent in the 8x8. Crossover was also fairly rare in the 8x8 direct solutions. The indexed 8x8 solutions, however, showed a fairly even distribution of the full set of variation operators. An example (simplified) solution from an 8x8 index run is shown in Figure 4. The program uses a cross-section of different variation operators, including an annealing mutation (the second *mutateX* expression). Note the high rate of mutation (48%). A variety of selection operators are used as well.

5 Discussion

Parameter	*Value*
Search set size	100
Runs/experiment	10
Replacement	tournament worst
Tournament size	2
Mutation rate	0.03
Annealing rate	0.9/iteration

Table 5
Common reference run parameters

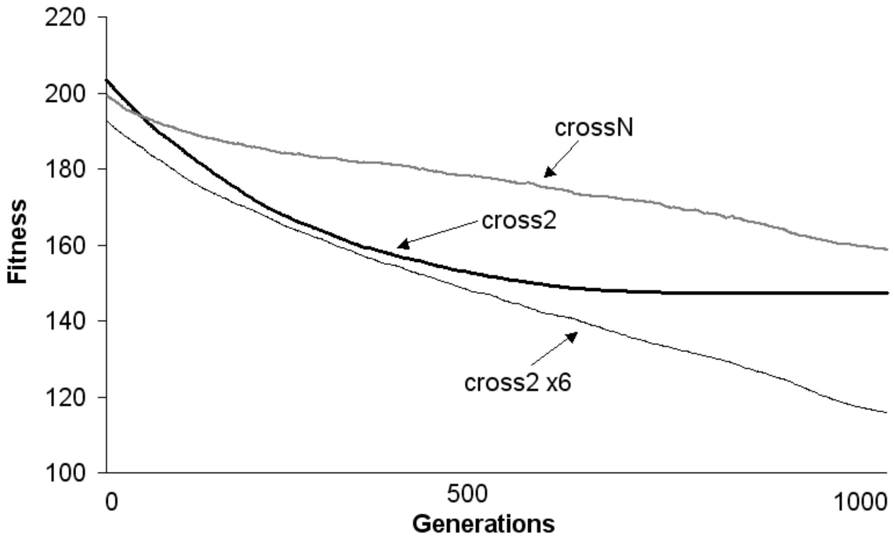

Fig. 5. Fitness performance comparison

As seen in Section 4, different styles of search algorithms were consistently derived for different search problems. To verify that search algorithms obtained are sensible for the problems used, some reference runs were done for a select sample of experiments, using some hand-coded search strategies. Some of the common parameters for these reference runs are shown in Table 5. The *cross* $\times$ 6 searches is the following memetic crossover structure:

```
(crossover
```

TSP	Variation operator	Max iteration	Selection	Score	Evolved algorithms
direct 8x8	mut	1000	best	288.6	
	mut/i	1000	best	⋆206.8	
	cross×6	1000	best, tourn	366.6	6 hill-climb,
	mut	1000	tourn	314.7	4 annealing
	mut/i	1000	tourn	275.5	
	cross×6	1000	tourn	331.0	
indexed 8x8	mut	1000	best	148.0	
	mut/i	1000	best	140.2	
	cross×6	1000	best, tourn	154.0	10 memetic
	mut	1000	tourn	142.4	
	mut/i	1000	tourn	136.8	
	cross×6	1000	tourn	⋆129.4	
direct 4x4	mut	3000	best	9.9	
	cross2×6	3000	best, tourn	20.0	
	cross2×6 mut×2	3000	best, tourn	11.4	10 memetic
	mut	3000	tourn	9.2	
	cross2×6	3000	tourn	11.6	
	cross2×6 mut×2	3000	tourn	⋆7.9	
indexed 8x8	cross2	1000	tourn	147.4	
	cross2×6	1000	tourn	⋆115.8	10 memetic
	crossN	1000	tourn	158.7	

Table 6
Reference run results

```
(crossover
      (crossover S S)
      (crossover S S))
(crossover
      (crossover S S)
      (crossover S S)))
```

where S is a selection operator. The $cross2 \times 6\ mut \times 2$ uses this expression:

```
(crossover2
      (crossover2
            (mutate 0.03
                  (crossover2 S S))
                  (crossover2 S S))
      (crossover2
             (mutate 0.03
                  (crossover2 S S))
                  (crossover2 S S)))
```

Also, note that applying a crossover to the same individual, as would be obtained with *sel_best*, results in no variation of that individual. Hence some runs combined *sel_best* with *sel_tourn_best* in order to obtain the possibility of creating variation.

Results of the reference runs are shown in Table 6. The score column refers to the average of the objective scores of the best solution over 10 runs. The single best result in each experiment is marked with a star. The evolved algorithms column recaps the information from Table 3. All these reference runs confirm that the style of search algorithm evolved is consistent with the relative performance of the search algorithm compared to other possibilities.

Table 6 highlights some pertinent points about the search strategies tested. Firstly, crossover-style variation operators do not perform well with hill-climbing selection (*sel_best*). Rather, crossover requires variability with the parent structures, and this is best done with probabilistic selection as done with tournament selection, or even with random selection (as was seen in many evolved algorithms). The mixture of mutation with crossover was found to be ideal in this regard, as can be seen by the better performance of the mixed crossover and mutation runs in the direct 4x4 experiment. As was evident in most evolved algorithms, crossover and mutation were often used together.

The indexed 8x8 runs highlight the fact that memetic crossover is a much more powerful reproduction operator than simple 2-parent crossover. Furthermore, contrary to intuition, memetic crossover is *not* equivalent to N-point crossover. The ability of memetic crossover to combine representation information from a wide variety of individuals in the search set is a decidedly powerful strategy. This can be seen in the performance graph in Figure 5, which is obtained by averaging the best scores obtained each generation during the reference runs. Note, however, that there is computational overhead involved in the memetic crossover, and this may be an issue in determining its practicality.

6 Conclusion

This research shows that search strategies are automatically derivable for search problems at the algorithmic level. Search algorithms for a variety of structured and unstructured TSP problems of varying complexity were evolved using genetic programming. Some analyses showed that the algorithms obtained were consistent with empirical performance of known search strategies for these problems.

There are a number of aspects of this research under continued investigation. The first issue is the degree of bias caused by the search language used. The language introduced in Section 3.2 is of limited scope and robustness, and is not intended to denote a large variety of search paradigms. The use of a fixed search set that can only grow greatly restricts the kinds of algorithms possible. The lack of more advanced data structures, such as stacks, is also limiting. Further work is under way to expand the generality of the search language. In particular, primitives are being written to enable tabu-style search.

Another bias in the implementation of the language is the set of primitives implemented, which are borrowed primarily from genetic algorithms. There are an enormous variety of variation operators in the literature. A more general language would include a larger set of such operators. Ideally, the details of the operators should themselves be evolvable, as done in [].

The genetic programming paradigm itself contributed some bias to the styles of algorithms obtained. Program bloat (intron, dead code) was prevalent in many runs. This was largely a result of the way the replacement and growth operators were implemented. For example, since only the final replacement operator would have an actual effect in the iteration step, the earlier replament operators can grown unfettered without affecting the overall result. Although sensible algorithms were nevertheless evolved, being able to restrict bloat would result in better performance overall in the search for algorithms. Bloat can also favour certain search operators over others. An expression like *(rep_tourn_worst E F)* is more likely to survive reproductive alteration than one like *(rep_worst F)*. This is because the tournament size term E in the first expression can grow very large, and hence protect the parent node from destruction. The *rep_worst* term does not have this genetic insurance policy.

By necessity, the search problems explored in our experiments were of limited complexity. The results we obtained in Sections 4 and 5 are limited by factors such as search set size, maximum iteration limit, and grid size. Therefore, these experimental decisions create a *search horizon*, and the results we obtained are valid for the extent of search accordingly undertaken. This point is important, as the analysis of the relative performance of various search algorithms in

Table 6 might be influenced by these experimental parameterizations. For example, our chosen maximum iteration limit of 1000 on a search set of 100 would be a very weak choice of parameters for a vanilla genetic algorithm run – it is equivalent to a mere 10 generations. More typically, a population of 500 might be run for 100 generations or more. The long-term performance characteristics of hill-climbing, annealing, or memetic crossover search on this extended search experiment might differ dramatically. Likewise, very different styles of search algorithms would likely evolve if such changes in parameters were influential.

This work contributes to the body of work in meta-heuristics and meta-evolution. One paper that inspired this research is that of Spector and Robinson, and their autoconstructive evolution experiments [15]. In that paper, they lay the foundation for evolving evolutionary operators and other behaviours, from the Push language – a general purpose stack-based language. Evolving evolutionary search from such a basic language is a much more ambitious goal that the evolution of search algorithms from higher-level search primitives. It is interesting to consider whether some level of abstraction between these two levels of meta-evolution might produce novel and powerful search paradigms heretofore unknown in the literature. For example, a more general-purpose search language might permit the synthesis of more complex problem-specific search strategies that coalesce the evalation function and variation operations. One might imagine giving the search language operators more details about the problem representation structure, and permitting the operators to compute a finer-resolution measurement of partial fitness. Such information could be used to derive much more effective search strategies, than is possible when fitness is evaluated as one monolithic value for an entire candidate solution, as is done in this paper.

Acknowledgement: This research is supported by NSERC Operating Grant 138467-1998.

References

[1] P.J. Angeline. Two Self-Adaptive Crossover Operators for Genetic Programming. In P.J. Angeline and K.E. Kinnear, editors, *Advances in Genetic Programming II*, pages 89–110. MIT Press, 1996.

[2] J.C. Culberson. On the Futility of Blind Search: An Algorithmic View of 'No Free Lunch'. *Evolutionary Computation*, 6(2):109–127, 1998.

[3] B. Edmonds. Meta-Genetic Programming: Co-evolving the Operators of Variation. *Electrik*, 9:13–29, 2001.

[4] T.C. Fogarty. Varying the Probability of Mutation in the Genetic Algorithm. In J. Shaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104–109. Morgan Kaufmann, 1989.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, 1979.

[6] G. Gutin and A.P. Punnen. *Traveling Salesman Problem and Its Variations.* Kluwer Academic Publishers, 2002.

[7] H. Iba and H. de Garis. Extending Genetic Programming with Recombinative Guidance. In P.J. Angeline and K.E. Kinnear, editors, *Advances in Genetic Programming II*, pages 69–88. MIT Press, 1996.

[8] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[9] N. Krasnogor. *Studies on the Theory and Design Space of Memetic Algorithms.* PhD thesis, Faculty of Computing, Engineering and Mathematical Sciences, University of the West of England, Bristol, 2002.

[10] K. H. Liang, X. Yao, and C. S. Newton. Adapting self-adaptive parameters in evolutionary algorithms. *Applied Intelligence*, 15(3):171–180, November/December 2001.

[11] Z. Michalewicz and D.B. Fogel. *How to Solve It: Modern Heuristics.* Springer Verlag, 2002.

[12] D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[13] B.M. Ombuki, M. Nakamura, and K. Onaga. An Evolutionary Scheduling Scheme Based on gkGA Approach to the Job Shop Scheduling Problem. *IEICE Trans. Fundamentals*, E81-A(6):1063–1071, June 1998.

[14] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[15] L. Spector and A. Robinson. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[16] H. Tamaki, H. Kita, N. Shimizu, K. Maekawa, and Y. Nishikawa. A Comparison Study of Genetic Codings for the Traveling Salesman Problem. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 1–6. IEEE Press, June 1994.

[17] A. Teller. Evolving Programmers: the Co-evolution of Intelligent Recombination Operators. In P. Angeline and K.E. Kinnear, editors, *Advances in Genetic Programming II*, pages 45–68. MIT Press, 1996.

[18] J.L. Wallis and S.K. Houghten. Comparative Study of Search Techniques Applied to the Minimum Distance Problem of BCH Codes. In *Proceedings 6th IASTED International Conference on Artificial Intelligence and Soft Computing*, pages 164–169, Banff, Alberta, July 2002.

[19] P. H. Winston. *Artificial Intelligence.* Addison Wesley, 1992.

[20] D.H. Wolpert and W.G. Macready. No Free Lunch Theorems for Search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, February 1996.

[21] D.H. Wolpert and W.G. Macready. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[22] D. Zongker and B. Punch. *lil-gp 1.0 User's Manual.* Dept. of Computer Science, Michigan State University, 1995.