

University of Montana

## ScholarWorks at University of Montana

---

Graduate Student Theses, Dissertations, &  
Professional Papers

Graduate School

---

1997

### Investigation into the search characteristics of three hillclimbing algorithms

David F. Glass  
*The University of Montana*

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

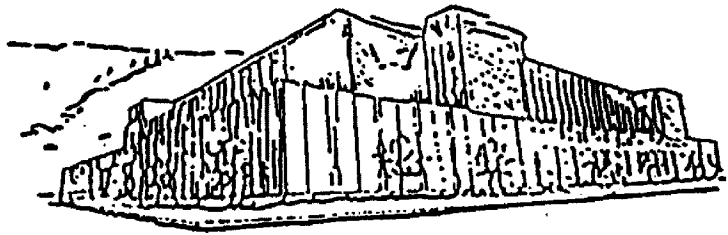
**Let us know how access to this document benefits you.**

---

#### Recommended Citation

Glass, David F., "Investigation into the search characteristics of three hillclimbing algorithms" (1997).  
*Graduate Student Theses, Dissertations, & Professional Papers*. 8189.  
<https://scholarworks.umt.edu/etd/8189>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact [scholarworks@mso.umt.edu](mailto:scholarworks@mso.umt.edu).



Maureen and Mike  
**MANSFIELD LIBRARY**

The University of **MONTANA**

---

Permission is granted by the author to reproduce this material in its entirety,  
provided that this material is used for scholarly purposes and is properly cited in  
published works and reports.

*\*\* Please check "Yes" or "No" and provide signature \*\**

Yes, I grant permission   X    
No, I do not grant permission       

Author's Signature

Date

11/15/97

Any copying for commercial purposes or financial gain may be undertaken only with  
the author's explicit consent.



AN INVESTIGATION INTO THE SEARCH CHARACTERISTICS  
OF THREE HILLCLIMBING ALGORITHMS

by

David F. Glass

B.A., The University of Montana, 1990

presented in partial fulfillment of the requirements

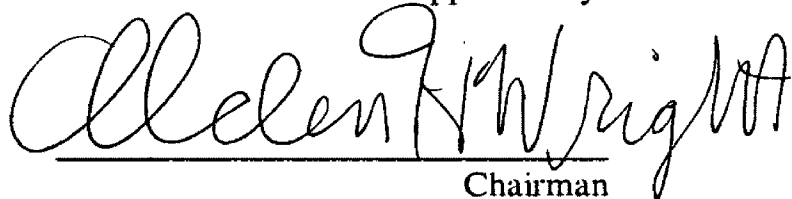
for the degree of


Master of Science

The University of Montana

December, 1997

Approved by:

  
Chairman

  
Dean, Graduate School

12-15-97

Date

UMI Number: EP38990

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP38990

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

**An Investigation into the Search Characteristics of Hillclimbing Algorithms (70 pp.)**

**Director: Alden H. Wright**



In recent years, iterative search techniques have emerged as practical and robust function optimization methods. The general method called hillclimbing is presented here. In particular, three hillclimbing algorithms that make use of the mutation operator are investigated. These are a single-bit mutation algorithm, *RMHC*, and two varieties of a multi-bit mutation (macromutation) algorithm, which we call *MMHC1* and *MMHC2*. Several important algorithm parameter settings are varied across the runs of these algorithms, and three different problem sizes are used. Each algorithm is tried on the six bit fully easy and six bit fully deceptive problems of Goldberg[7] as well as on the *NK* landscapes due to Kauffman[5].

The results indicate the relative superiority of the macromutation algorithms as compared with the single-bit mutation algorithm. This is especially true for the macromutation algorithm, *MMHC2*, whose mutated bits are localized with respect to each other along the length of the bit string. Also, it is seen in each problem that as the number of bits mutated is increased past a certain value, performance of all the algorithms is degraded. The effect on relative algorithm performance of variations in other parameter settings is also illustrated. An answer to the question, "Why does macromutation do well on separable functions?" is given and the effect of problem length on algorithm performance is examined.

# Table of Contents

ABSTRACT.....	ii
TABLE OF CONTENTS.....	1
LIST OF FIGURES .....	3
CHAPTER 1: INTRODUCTION TO HILLCLIMBING .....	4
1.1 A Problem .....	4
1.2 A Solution .....	5
1.3 Benefits .....	7
CHAPTER 2: THE ALGORITHMS .....	8
2.1 The Algorithms.....	8
2.1.1 General Hillclimbing Algorithm.....	8
2.1.2 <i>RMHC</i> .....	11
2.1.3 <i>MMHC1</i> .....	11
2.1.4 <i>MMHC2</i> .....	12
2.2 Linear (affine) Interaction, Coupling and Separability .....	13
2.2.1 Linear (affine) Interaction and Coupling .....	13
2.2.2 Separability.....	16
2.3 The Hypotheses .....	17
2.3.1 Macromutation .....	17
2.3.1.1 Overall Performance .....	17
2.3.1.2 Number and Position of Bits Mutated.....	17
2.3.1.3 <i>RMHC</i> 's Strength .....	17
2.3.2 <i>emoves</i> .....	18
2.3.3 <i>nmoves</i> .....	18
2.3.4 Informal Summary of Hypotheses and Observations .....	19
CHAPTER 3: THE PROBLEMS.....	20
3.1 Fully Easy .....	21
3.2 Fully Deceptive .....	22
3.3 <i>NK</i> Landscapes .....	23
3.4 Formal Statement of Hypotheses and Observations .....	26
CHAPTER 4: THE EXPERIMENTS .....	27
4.1 Parameters.....	27
4.1.1 <i>N</i> .....	27
4.1.2 <i>nmoves</i> and <i>emoves</i> .....	28
4.1.3 <i>bits</i> .....	28
4.1.4 <i>K</i> .....	29
4.2 Procedures.....	30
4.2.1 Implementation.....	30
4.2.2 Preliminary Hillclimbs.....	30
4.2.3 Final Hillclimbs .....	31

<b>CHAPTER 5: DATA AND RESULTS</b> .....	<b>33</b>
5.1 All Algorithms / All Problem Sizes .....	33
5.1.1 All Algorithms: bits mutated .....	33
5.1.1.1 Deceptive.....	33
5.1.1.2 Easy and $NK$ .....	34
5.1.2 All Algorithms: bits mutated / $n$ moves and $e$ moves .....	35
5.1.2.1 Deceptive.....	35
5.1.2.2 Easy.....	36
5.1.2.3 $NK$ Landscapes.....	37
5.1.3 All Algorithms: $K$ .....	38
5.1.3.1 $NK$ Landscapes .....	38
5.1.4 All Algorithms: $K$ / $n$ moves and $e$ moves .....	39
5.1.4.1 $NK$ Landscapes .....	39
5.2 $MMHC1$ / All Problem Sizes.....	40
5.2.1 $MMHC1$ / Deceptive: bits mutated .....	40
5.2.2 $MMHC1$ / Deceptive: bits mutated / $n$ moves and $e$ moves .....	41
5.2.3 $MMHC1$ / Deceptive: $N$ / $n$ moves and $e$ moves .....	42
5.2.4 $MMHC1$ / Easy: bits mutated .....	43
5.2.5 $MMHC1$ / Easy: bits mutated / $n$ moves and $e$ moves .....	44
5.2.6 $MMHC1$ / Easy: $N$ / $n$ moves and $e$ moves .....	45
5.2.7 $MMHC1$ / $NK$ Landscapes: bits mutated / $K$ .....	46
5.2.8 $MMHC1$ / $NK$ Landscapes: bits mutated / $n$ moves and $e$ moves.....	47
5.2.9 $MMHC1$ / $NK$ Landscapes: $K$ / $n$ moves and $e$ moves .....	48
5.3 $MMHC2$ / All Problem Sizes.....	49
5.3.1 $MMHC2$ / Deceptive: bits mutated .....	49
5.3.2 $MMHC2$ / Deceptive: bits mutated / $n$ moves and $e$ moves .....	50
5.3.3 $MMHC2$ / Deceptive: $N$ / $n$ moves and $e$ moves .....	51
5.3.4 $MMHC2$ / Easy: bits mutated .....	52
5.3.5 $MMHC2$ / Easy: bits mutated / $n$ moves and $e$ moves .....	53
5.3.6 $MMHC2$ / Easy: $N$ / $n$ moves and $e$ moves .....	54
5.3.7 $MMHC2$ / $NK$ Landscapes: bits mutated / $K$ .....	55
5.3.8 $MMHC2$ / $NK$ Landscapes: bits mutated / $n$ moves and $e$ moves.....	56
5.3.9 $MMHC2$ / $NK$ Landscapes: $K$ / $n$ moves and $e$ moves .....	57
5.4 $RMHC$ / All Problem Sizes .....	58
5.4.1 $RMHC$ / Deceptive: $bits = 1$ / $n$ moves and $e$ moves .....	58
5.4.2 $RMHC$ / Easy: $bits = 1$ / $n$ moves and $e$ moves .....	59
5.4.3 $RMHC$ / $NK$ Landscapes: $bits = 1$ / $n$ moves and $e$ moves .....	60
<b>CHAPTER 6: CONCLUSIONS</b> .....	<b>61</b>
6.1 Superiority of Macromutation.....	61
6.2 Superiority of Localized Macromutation.....	62
6.3 The Strength of $RMHC$ .....	63
6.4 The effects of varying $n$ moves, $e$ moves and $N$ .....	64
6.4.1 $n$ moves and $e$ moves .....	64
6.4.2 $N$ .....	65
6.4.3 The $bits$ Parameter .....	66
6.5 Summary .....	66
<b>APPENDIX</b> .....	<b>68</b>
<b>BIBLIOGRAPHY</b> .....	<b>70</b>



# List of Figures

<i>Figure 2.1. Pseudo-code for the general hillclimbing algorithm.</i>	10
<i>Figure 2.2. Pseudo-code for mutation under RMHC.</i>	11
<i>Figure 2.3. Pseudo-code for mutation under MMHC1.</i>	12
<i>Figure 2.4. Pseudo-code for mutation under MMHC2.</i>	12
<i>Figure 3.1. A fully easy six bit problem. Maximum fitness point is 000000.</i>	21
<i>Figure 3.2. A fully deceptive six bit problem. Maximum fitness point is 111111.</i>	22
<i>Figure 3.3. Elements of an NK function where <math>N = 6</math> and <math>K = 2</math> (substring length = 3).</i>	25
<i>Figure 3.4. NK Fitness calculation for the string 101100.</i>	25
<i>Figure 4.1. Summary of elements comprising the hillclimbs.</i>	29
<i>Figure 5.1 – 5.48. Charts graphically showing hillclimb results.</i>	33 – 60
<i>Figure A.1. Sample UNIX C-Shell script used to run the MMHC2 algorithm.</i>	68
<i>Figure A.2. Sample UNIX command-line invocation of the MMHC1 algorithm.</i>	69
<i>Figure A.3. Sample UNIX command-line invocation of the MMHC2 algorithm.</i>	69

# Chapter 1

## Introduction to Hillclimbing

### 1.1 A Problem

Real world problems whose instances contain many variables pose a daunting challenge to people charged with arriving at optimal solutions to those problems in a reasonable amount of time, using a reasonable amount of resources. One such problem might be the weekly scheduling of employee work shifts at a large company. Company management would have to take into account the various requirements of each of the many jobs involved in their business and the suitability of the qualifications of the employees available to fill those jobs. The employees' personal preferences regarding the shifts they worked, as well as the demands placed on them by their outside responsibilities would all be factors influencing such a decision. Other considerations, such as the completion times allotted for necessary tasks, the average rates at which the individual employees have worked in the past to complete such tasks, job deadlines and even the availability of employees due to holidays are examples of the many additional factors complicating cost

effective arrival at an optimal solution. An optimal solution in this case would likely be one that satisfied the most employees while at the same time allowing completion of the most work in the least amount of time. One method of searching for solutions to such difficult and multifaceted problems would be to work out with pencil and paper each different combination of employees in different shifts, evaluating each potential work schedule, one at a time, according to its overall effectiveness, given the criteria mentioned above. For a small company of five employees, this method of work shift scheduling might be sufficient. For a company of 200 employees, this weekly task would be prohibitively complex.

## **1.2 A Solution**

As the size and complexity of multivariate optimization problems like this one grow, iterative techniques that use various operators to navigate the search space become viable alternatives to exhaustive search[2]. If we are able to find a representation for a particular problem that easily fits into a familiar, expedient solution technique, then we have accomplished a large part of the task of efficiently searching for better solutions to our problems. By way of example, using the particular problem mentioned, one combination of employees scheduled to work certain shifts might be better than another combination. We might assign a numeric value (fitness) to each of the two schedules, depending on how good they are, given the criteria we have chosen to use to evaluate them. Changing one employee's shift assignment might be similar to complementing one bit in a bit string. It would likely have some effect (small or profound) on the relative value of that

particular schedule. Trying different combinations of shift schedules would be similar to mutating bits in a bit string, thus altering string fitness.

Representing multivariate problems as bit strings allows us to apply computerized search methods to them. One such computerized search technique is the iterative procedure called hillclimbing[1]. We perform a hillclimb on a bit string simply by repetitively changing bits in the string, which hopefully results in our finding a better bit string, one step at a time[4]. Before we perform a hillclimb on a bit string though, we must have a place to start. In this work, we start at a point, the composition of which determines its position on a landscape. Each of the points on the landscape consists of a concatenation of  $N$  bits (a string of length  $N$ ), with each bit valued at 0 or 1. The landscape itself consists of  $2^N$  of these points. Associated with each point on the landscape is a real-valued number between 0 and 1 that depends on the values of the point's constituent bits and their locations in the string. This number is called the fitness of the point. (Fitness can be thought of as the characteristic of a point that determines its place among all of the  $2^N$  points comprising the landscape.) We conduct our search for highly fit points in this search space. In so doing, we are moving on the landscape. The immediate neighbors of any point in the search space are points that differ from it in the value of only one bit (in the case of the *RMHC* algorithm) and up to as many as six bits (in the *MMHC1* and *MMHC2* algorithms). We randomly choose a point on the landscape and call it the current point. We immediately complement one or more of the point's bits, effectively changing its location on the landscape to that of one of its neighbors. We evaluate the new point's fitness. At this point, there are three possibilities. If the point newly created

by mutation is of higher fitness than the previous point, we accept it as our current point and continue from there. If the new point is of lower fitness than the previous point, we apply mutation to the previous point again, generating another new string to evaluate. We only do this up to a specified number of times (*nmoves*), after which time a new random starting point is generated. If the new point is of equal fitness to the previous point, we accept it as our new current point. This also, is done up to a specified number of times (*emoves*), after which time a new random starting point is generated. This mutate-evaluate-select procedure is iterated until there is no additional fitness improvement, within the limits set by the *nmoves* and *emoves* parameters. In the case of our study, we also limit this process by ending it after a fixed number of fitness evaluations.

### 1.3 Benefits

The object of hillclimbing is to locate points of high fitness in the search space. By representing the multiple facets of some real-world scenario in the form of a binary string, we can effectively map a complicated multivariate problem to a form that can be efficiently explored in search of improved solutions using a computer. We examine here the effects of adjusting some of the parameters involved in applying each of three hillclimbing algorithms to bit strings of three different lengths.

# Chapter 2

## The Algorithms

### 2.1 The Algorithms

Initially, we present the general algorithm used for hillclimbing. (See Figure 2.1.) This will be followed by a detailed description of the characteristics that differentiate each of the three varieties of this general algorithm. As it turns out, *RMHC*, *MMHCI* and *MMHC2* differ only in the procedure that performs mutation on the bit strings.

#### 2.1.1 General Hillclimbing Algorithm

In this study, we perform a fixed number of function evaluations for each combination of algorithm parameter settings as a way of allocating an equal amount of work to each algorithm. For this purpose, we define a maximum number of evaluations called *maxevals*. We begin the general hillclimbing algorithm by using a random number generator to generate a random point in the search space. (For example, if it is a 60 bit

problem that is being used in this hillclimb, then we start with a string of 60 randomly generated bits.) Call this point *currentPoint*. We evaluate that string's fitness by using the fitness function we are studying. Call that number *currentFitness*. The next step is to apply a mutation operator to the string. This creates *newPoint*. The mutation operator is a function that is called by the algorithm. (See the discussion concerning the separate mutation operators, in sections 2.1.2, 2.1.3 and 2.1.4.) The fitness of *newPoint* is evaluated. Call that number, *newFitness*. Depending on the outcome of a comparison of *newFitness* with *currentFitness*, one of three possible paths is taken through the algorithm:

- ◆ ***newFitness* > *currentFitness***: We accept the newly mutated string as our new current point and repeat the mutate-evaluate-select process from there.
- ◆ ***newFitness* < *currentFitness***: We increment a variable, *ntimes*, that keeps track of how many times a newly mutated point is less fit than the current point. We check the *nmoves* parameter. If *ntimes* is less than the *nmoves* parameter, then we repeat the mutate-evaluate-select process on the current string, the one that had originally resulted in *currentFitness*. If *ntimes* is greater than or equal to the *nmoves* parameter, we consider that we have reached a point of locally maximum fitness on the landscape and start a fresh hillclimb. If the number of function evaluations performed has reached or exceeded *maxevals*, we terminate that set of hillclimbs, return the point of highest fitness achieved and begin a new set of hillclimbs using a new combination of parameter settings.
- ◆ ***newFitness* = *currentFitness***: We increment a variable, *etimes*, that keeps track of how many times a newly mutated point is equal in fitness to the current point. We check the *emoves* parameter. If *etimes* is less than the *emoves* parameter, we accept the new, equally fit point as our new current point. We continue the hillclimb from there by repeating the mutate-evaluate-select process. If *etimes* is greater than or equal to the *emoves* parameter, we consider that we have reached a point of locally maximum fitness on the landscape and start a new hillclimb. If the number of function evaluations performed has reached or exceeded *maxevals*, we terminate that set of hillclimbs, return the point of highest fitness achieved and begin a new set of hillclimbs using a new combination of parameter settings.

**Figure 2.1.** Pseudo-code for the general hillclimbing algorithm.

```

HILLCLIMB(maxevals, nmoves, emoves)
  evals  $\leftarrow$  superfit  $\leftarrow$  0
  while evals < maxevals
    ntimes  $\leftarrow$  etimes  $\leftarrow$  0
    climb  $\leftarrow$  'yes'
    currentPoint  $\leftarrow$  generateRandomPoint()
    currentFitness  $\leftarrow$  evaluateFitness(currentPoint)
    evals  $\leftarrow$  evals + 1
    while climb = 'yes'
      newPoint  $\leftarrow$  mutate(currentPoint)
      newFitness  $\leftarrow$  evaluateFitness(newPoint)
      evals  $\leftarrow$  evals + 1

      if newFitness > currentFitness
        then ntimes  $\leftarrow$  etimes  $\leftarrow$  0
             currentPoint  $\leftarrow$  newPoint
             currentFitness  $\leftarrow$  newFitness

      else if newFitness < currentFitness
        then ntimes  $\leftarrow$  ntimes + 1
             if ntimes  $\geq$  nmoves
               then climb  $\leftarrow$  'no'

      else if newFitness = currentFitness
        then etimes  $\leftarrow$  etimes + 1
             ntimes  $\leftarrow$  0
             if etimes  $\geq$  emoves
               then climb  $\leftarrow$  'no'
             else currentPoint  $\leftarrow$  newPoint
                  currentFitness  $\leftarrow$  newFitness

    if evals  $\geq$  maxevals
      then climb  $\leftarrow$  'no'

  if currentFitness > superfit
    then superfit  $\leftarrow$  currentFitness

return superfit

```



### 2.1.2 *RMHC*

The Random Mutation Hillclimbing algorithm is the simplest of the three algorithms. In the mutation step of *RMHC*, we mutate one bit chosen at random from the  $N$  bits of the string. That is, we change its value to that of its complement. (See Figure 2.2.)

**Figure 2.2.** Pseudo-code for mutation under *RMHC*.

```

RMHC_MUTATE(string, $N$ )
   $r \leftarrow \text{generateRandomInteger}(0 : N - 1)$ 
  complement(string[ $r$ ])

  return string

```

### 2.1.3 *MMHC1*

In the Macro-Mutation Hillclimbing algorithm number 1 (*MMHC1*), when we apply the mutation step, instead of mutating only one bit (as in *RMHC*), we mutate some number of bits in the string according to the result of a comparison. The comparison is between  $r$ , chosen uniformly at random from the interval  $(0, 1)$ , and the quotient,  $\text{bits}/N$  (where *bits* is the *bits* parameter the algorithm is using and  $N$  is the length of the string whose bits are being mutated). Starting with the first bit in the string to be mutated, and proceeding to the last, for each bit, we generate  $r$  and we mutate that bit if  $r < (\text{bits}/N)$ . This formula for mutating bits insures that the mutated bits are distributed randomly throughout the string (See Figure 2.3.)

**Figure 2.3.** Pseudo-code for mutation under *MMHC1*.

```

MMHC1_MUTATE(string,bits,N)
  for  $i \leftarrow 0$  to  $(N - 1)$ 
    do  $r \leftarrow \text{generateRandomReal}(0 : 1)$ 
      if  $r < (\text{bits} / N)$ 
        then complement(string[ $i$ ])

  return string

```

#### 2.1.4 *MMHC2*

In the Macro-Mutation Hillclimbing algorithm number 2 (*MMHC2*), we mutate some of the bits in the string to be mutated according to a formula that insures that the mutated bits remain within a fixed distance of each other (localized) rather than occurring throughout the entire string (distributed), as in *MMHC1*. To mutate bits in *MMHC2*, we first generate a random starting position in the  $N$  bits of the string. Letting  $m = 2 * \text{bits}$ , where *bits* is the algorithm's *bits* parameter, for each of the following  $m$  bits from the starting position, we generate a real-valued, random number,  $r$ , with a uniform distribution between 0 and 1. We mutate the bit in question if  $r < 0.5$ . (See Figure 2.4)

**Figure 2.4.** Pseudo-code for mutation under *MMHC2*.

```

MMHC2_MUTATE(string,bits,N)
  start  $\leftarrow \text{generateRandomInteger}(0 : N - 1)$ 
   $m \leftarrow 2 * \text{bits}$ 
  for  $i \leftarrow \text{start}$  to  $(\text{start} + m - 1)$ 
    do  $r \leftarrow \text{generateRandomReal}(0 : 1)$ 
      if  $r < 0.5$ 
        then complement(string[ $i \bmod N$ ])

  return string

```

## 2.2 Linear (Affine) Interaction, Coupling and Separability

### 2.2.1 Linear (Affine) Interaction and Coupling

We would like to show what it means for a function to be affine in two of its bits.

Consider two bits,  $x_0$  and  $x_1$  such that  $x_0, x_1 \in \{0, 1\}$ . A function of two bits,  $g(x_0, x_1)$ , is affine if there exist real valued constants,  $a, b, c$ , such that  $g(x_0, x_1) = ax_0 + bx_1 + c$ .

**Proposition:** A function of two bits,  $x_0$ , and  $x_1$ ,  $g(x_0, x_1)$ , is affine if and only if

$$g(1, 1) - g(0, 1) = g(1, 0) - g(0, 0).$$

**Proof:**

$$\Rightarrow \quad g(1, 1) = (a \cdot 1) + (b \cdot 1) + c = (a + b + c)$$

$$g(0, 1) = (a \cdot 0) + (b \cdot 1) + c = (b + c)$$

$$g(1, 0) = (a \cdot 1) + (b \cdot 0) + c = (a + c)$$

$$g(0, 0) = (a \cdot 0) + (b \cdot 0) + c = c$$

$$\Leftarrow \quad \text{Let } a = g(1, 0) - g(0, 0), b = g(0, 1) - g(0, 0), \text{ and } c = g(0, 0).$$

$$\begin{aligned} g(1, 1) &= [g(1, 0) - g(0, 0)] \cdot 1 + [g(0, 1) - g(0, 0)] \cdot 1 + g(0, 0) \\ &= (a \cdot 1) + (b \cdot 1) + c \end{aligned}$$

$$\begin{aligned} g(0, 1) &= [g(1, 0) - g(0, 0)] \cdot 0 + [g(0, 1) - g(0, 0)] \cdot 1 + g(0, 0) \\ &= (a \cdot 0) + (b \cdot 1) + c \end{aligned}$$

$$\begin{aligned} g(1, 0) &= [g(1, 0) - g(0, 0)] \cdot 1 + [g(0, 1) - g(0, 0)] \cdot 0 + g(0, 0) \\ &= (a \cdot 1) + (b \cdot 0) + c \end{aligned}$$

$$\begin{aligned} g(0, 0) &= [g(1, 0) - g(0, 0)] \cdot 0 + [g(0, 1) - g(0, 0)] \cdot 0 + g(0, 0) \\ &= (a \cdot 0) + (b \cdot 0) + c \end{aligned} \quad \blacksquare$$

A fitness function,  $F(x_0, \dots, x_{N-1})$  is affine in bits  $x_i$  and  $x_j$  if, for any choice of the remaining bits  $x_k, k \neq i, j$ ,

$$F(x_0, \dots, x_{i-1}, 1, \dots, x_{j-1}, 1, \dots, x_{N-1}) - F(x_0, \dots, x_{i-1}, 0, \dots, x_{j-1}, 1, \dots, x_{N-1}) = \\ F(x_0, \dots, x_{i-1}, 1, \dots, x_{j-1}, 0, \dots, x_{N-1}) - F(x_0, \dots, x_{i-1}, 0, \dots, x_{j-1}, 0, \dots, x_{N-1}).$$

In other words,  $F$  is affine in bits  $x_i$  and  $x_j$  if  $g(x_i, x_j) = F(x_0, \dots, x_{i-1}, x_i, \dots, x_{j-1}, x_j, \dots, x_{N-1})$  is affine. If  $F$  is nonaffine in  $x_i$  and  $x_j$ , then we say  $x_i$  and  $x_j$  are coupled in  $F$ .

For example, we define a fitness function,  $F$ , as the sum of simpler, nonaffine functions,  $G_0, G_1, G_2$ , and  $G_3$ , and show what it means for  $F$  to be affine in two of its bits, say  $x_1$  and  $x_3$ . Let  $F(x_0, x_1, x_2, x_3) = G_0(x_0, x_1) + G_1(x_1, x_2) + G_2(x_2, x_3) + G_3(x_3, x_0)$ . The following table defines  $G_0, G_1, G_2$ , and  $G_3$ :

	$G_0$	$G_1$	$G_2$	$G_3$
00	2	6	3	9
01	3	0	2	9
10	1	4	1	6
11	0	9	8	4

From this, we can compute the following table of values for  $F$ :

		$x_2x_3$			
		00	01	10	11
$x_0x_1$	00	20	16	12	16
	01	19	15	22	26
	10	19	13	11	13
	11	16	10	19	11

e.g.,  $F(0\ 0\ 1\ 0) = 12$

We show  $F$  to be affine in  $x_1$  and  $x_3$ , holding bits  $x_0$  and  $x_2$  fixed.

Case 1:  $x_0 = 0, x_2 = 0$ . Let  $g(x_1, x_3) = F(0, x_1, 0, x_3)$ :

$$\begin{aligned} F(0\ 1\ 0\ 1) - F(0\ 0\ 0\ 1) &= F(0\ 1\ 0\ 0) - F(0\ 0\ 0\ 0) \Leftrightarrow \\ (3 + 4 + 2 + 6) - (2 + 6 + 2 + 6) &= (3 + 4 + 3 + 9) - (2 + 6 + 3 + 9) \Leftrightarrow \\ 15 - 16 &= 19 - 20 \Leftrightarrow -1 = -1 \end{aligned}$$

Case 2:  $x_0 = 0, x_2 = 1$ . Let  $g(x_1, x_3) = F(0, x_1, 1, x_3)$ :

$$\begin{aligned} F(0\ 1\ 1\ 1) - F(0\ 0\ 1\ 1) &= F(0\ 1\ 1\ 0) - F(0\ 0\ 1\ 0) \Leftrightarrow \\ 26 - 16 &= 22 - 12 \Leftrightarrow 0 = 0 \end{aligned}$$

Case 3:  $x_0 = 1, x_2 = 0$ . Let  $g(x_1, x_3) = F(1, x_1, 0, x_3)$ :

$$\begin{aligned} F(1\ 1\ 0\ 1) - F(1\ 0\ 0\ 1) &= F(1\ 1\ 0\ 0) - F(1\ 0\ 0\ 0) \Leftrightarrow \\ 10 - 13 &= 16 - 19 \Leftrightarrow -3 = -3 \end{aligned}$$

Case 4:  $x_0 = 1, x_2 = 1$ . Let  $g(x_1, x_3) = F(1, x_1, 1, x_3)$ :

$$\begin{aligned} F(1\ 1\ 1\ 1) - F(1\ 0\ 1\ 1) &= F(1\ 1\ 1\ 0) - F(1\ 0\ 1\ 0) \Leftrightarrow \\ 21 - 13 &= 19 - 11 \Leftrightarrow 8 = 8 \end{aligned}$$

We show  $F$  to be nonaffine in bits  $x_2$  and  $x_3$ , holding bits  $x_0$  and  $x_1$  fixed.

Let  $x_0 = 0, x_1 = 0, g(x_2, x_3) = F(0, 0, x_2, x_3)$ :

$$\begin{aligned} F(0\ 0\ 1\ 1) - F(0\ 0\ 0\ 1) &\neq F(0\ 0\ 1\ 0) - F(0\ 0\ 0\ 0) \Leftrightarrow \\ (2 + 0 + 8 + 6) - (2 + 6 + 2 + 6) &\neq (2 + 0 + 1 + 9) - (2 + 6 + 3 + 9) \Leftrightarrow \\ 16 - 16 &\neq 12 - 20 \Leftrightarrow 0 \neq -8 \end{aligned}$$

We say bits  $x_2$  and  $x_3$  are coupled in  $F$ .

As we have stated, *MMHC1* distributes its mutated bits across the string length,  $N$ . The local algorithm (*MMHC2*) mutates bits that are positionally close. This positional difference allows us to observe the effect of varying the coupling of the bits we mutate. For the test functions we use, the fitness of an entire string is made up of a sum of fitness functions that depend on substrings. (In the previous example,  $F$  was defined as the sum of  $G_0$ ,  $G_1$ ,  $G_2$ , and  $G_3$ , where each  $G_i$  depends on the values of two adjacent bits that comprise a two-bit substring.) In our test functions, if bit  $i$  and bit  $j$  are in the same substring, then they are coupled in the fitness function. Conversely, in our test functions, bits that are not in the same substring are not coupled in the fitness function. In this case, mutating one bit does not affect the fitness of the substring containing the other. For our test functions, bit  $i$  and bit  $j$  are affine in the fitness function if they are greater than  $k$  bits apart,

$$|i - j| > k,$$

where  $k$  is the length of the substrings used in the definition of the test function.

### 2.2.2 Separability

As stated, for the test functions we use, the fitness of an entire string is made up a sum of fitnesses of substrings. If the bits contributing to the fitness of a substring do not overlap with bits in other substrings, we say that the function is separable. In other words,  $F$  is separable if, for some  $i$ , and some functions  $G, H$ ,

$$F(x_0, \dots, x_{i-1}, x_i, \dots, x_{N-1}) = G(x_0, \dots, x_{i-1}) + H(x_i, \dots, x_{N-1}).$$

In this case, if  $j < i$  and  $i \leq k$ , then bits  $x_j$  and  $x_k$  are affine in  $F$ .

## **2.3 The Hypotheses**

### **2.3.1 Macromutation**

#### **2.3.1.1 Overall Performance**

We hypothesize that the two macromutation algorithms, *MMHC1* and *MMHC2*, perform better overall than the single-bit-flipping *RMHC* algorithm. We expect this to be true on the landscapes that have non-separable functions.

#### **2.3.1.2 Number and Position of Bits Mutated**

When hillclimbing algorithms mutate bits in a string in an attempt to find a point of high fitness, they can mutate one or several bits in any single iteration. The *RMHC* algorithm we study mutates only one bit per iteration while the *MMHC1* and *MMHC2* algorithms mutate multiple bits in an iteration. The number and position of bits mutated, especially as related to each other, are extremely important factors. We conjecture that mutating bits that are likely to be affine in the fitness function (*MMHC1*) results in somewhat lower performance levels on all test problems than mutating bits that are likely to be coupled in the fitness function (*MMHC2*).

#### **2.3.1.3 RMHC's Strength**

We expect *RMHC*'s performance on the separable problems to be somewhat better than its performance on the non-separable problems.

### 2.3.2 *moves*

In hillclimbing algorithms, we are always searching for points (strings) that have higher fitness than the point where we currently are. If every mutation resulted in a point of higher fitness, every hillclimb would be simply a matter of directly taking the shortest route to the top of the highest peak on the landscape. Of course, that is what we hope will happen. The random nature of the mutation operator seldom affords such a fortuitous series of steps. Often, we find ourselves at relatively flat places on a landscape (mesas) where most mutations result in points of equal fitness. Although the moves of hillclimbing algorithms are biased in the direction of strings of higher fitness, we hypothesize that it would be advantageous to allow the algorithms to make moves to points of equal fitness in hopes of discovering higher-fitness points hidden on the mesa. Therefore, we have implemented an *moves* parameter, which is some multiple of the string length,  $N$ . We investigate the benefit of the *moves* parameter to the performance of the algorithms and observe the effect of its becoming quite large with respect to  $N$ .

### 2.3.3 *nmoves*

The *moves* parameter, discussed above, is one way that we limit possibly endless wandering of our algorithms on non-productive parts of the landscapes. The *nmoves* parameter is another way to accomplish a similar end. When a mutation results in a less fit string, we throw away the original string in favor of a fresh hillclimb only after we have tried what we have decided to allow as a sufficient number of mutations of the original string. However, this state of affairs might also indicate that we have reached a point of locally maximum fitness. The parameter that determines the number of lower-



fitness mutations that we consider before we make that decision is the *nmoves* parameter. We think that allowing an increase in *nmoves* would be beneficial to algorithm performance. As with *moves*, we set *nmoves* to a multiple of  $N$ . While we think that, like *moves*, *nmoves* should increase, we investigate the effect of constraining its upper limit to a smaller multiple of  $N$  than that for *moves*.

### 2.3.4 Informal Summary of Hypotheses and Observations

- *MMHC1* and *MMHC2* perform better than *RMHC* on all test problems and on all problem sizes. This is demonstrated emphatically on the problems using non-separable functions.
- Mutating bits that are likely to be affine in the fitness function (*MMHC1*) results in somewhat lower performance levels on all test problems than mutating bits that are likely to be strongly coupled in the fitness function (*MMHC2*).
- *RMHC* performs better on the problems using separable functions than on the problems using non-separable functions, for all problem sizes.
- We observe the effect on algorithm performance of the *moves* and *nmoves* parameters.
- We observe the effect of the problem length,  $N$ , on algorithm performance.

## Chapter 3

### The Problems

It seems clear enough why we would want to automate the solution of a problem like the one presented in Chapter 1: time and money. Of course, the method chosen to implement the automation should be as efficient as the current state of technology allows. The hillclimbing algorithms that we explore here present areas into which investigation can result in increased understanding of the method and allow for adjustment of the parameters that will result in optimal algorithmic performance.

The landscapes (problems) we have chosen for our hillclimbing investigation are the six bit fully easy and six bit fully deceptive landscapes found in Goldberg[7] and the *NK* landscapes due to Kauffman[5].

### 3.1 Fully Easy

This problem requires that the algorithm simultaneously solve a number of fully easy subproblems. Each subproblem is a six bit function of unitation. Unitation is the number of bits in a binary string that are set to 1. (The specific function of unitation is given in Figure 3.1.) For example, the string, *010101* contains three ones. Thus its fitness would be 0.9. This landscape was designed by Goldberg[7] in his study of GAs. He intended it to be easy for a GA to solve. That is, it should be easy for that algorithm to achieve the maximum fitness point, the string of all zeroes (fitness = 1.0), in a fully easy problem. Of course, a hillclimbing algorithm is not a GA and will find other local maxima on this landscape besides the global one. However, the fully easy landscape is one that is useful for comparing the three algorithms that we have chosen for this work because it is a separable function. Each problem instance is constructed by concatenating some number of these six bit functions together. The experiments use 10, 20 and 30 such subproblems, resulting in problem sizes of 60, 120 and 180 bits.

**Figure 3.1.** A fully easy six bit problem. Maximum fitness point is *000000*.

Unitation	0	1	2	3	4	5	6
Fitness	1.0	0.8	0.6	0.9	0.5	0.7	0.9

### 3.2 Fully Deceptive

The fully deceptive problems are also representative of a class of problems that have received attention in the study of GAs. Each of the fully deceptive subproblems is a six bit function of unitation. The string, *010101* here has fitness 0.3. (See Figure 3.2.) The function values for the different unitation values are arranged differently than in the fully easy problems. The maximum fitness point in a fully deceptive problem is the string of all ones. It is important to note that the location of the global maximum in the fully deceptive problem is opposite to the location of the global maximum in the fully easy problem (the string of all ones as opposed to the string of all zeroes). On the fully deceptive landscape, there is a local maximum situated at the point of all zeroes. Fully deceptive problems get their name from reasoning by their designer that the placement of function values would deceive an algorithm into finding the local maximum rather than the global maximum. As with the easy functions, deceptive functions are also separable functions whose bits are affine in the fitness function. Each of our problem instances is constructed by concatenating some number of these six bit functions together. The experiments use 10, 20 and 30 such subproblems, resulting in problem sizes of 60, 120 and 180 bits.

**Figure 3.2.** A fully deceptive six bit problem. Maximum fitness point is *111111*.

Unitation	0	1	2	3	4	5	6
Fitness	0.90	0.45	0.35	0.30	0.30	0.25	1.00

### 3.3 *NK* Landscapes

In the *NK* model,  $N$  refers to the number of parts of a system – factors influencing a decision, bits in a string, or otherwise. Each part makes a fitness contribution to the whole which depends upon that part and upon  $K$  other parts among the  $N$ . That is,  $K$  reflects the degree to which the system components are coupled with each other, the degree of epistatic interaction. In terms of the bit strings we use here,  $N$  refers to the total length of the string and  $K$  refers to the length of a substring within the  $N$  bits[5,6].

For example, if we have a string of 60 bits ( $N = 60$ ), then there are ten substrings of length six ( $K = 5$ ) which comprise the 60 bits. This is true when the substrings are placed end-to-end. However, the substrings might not be placed end-to-end. The substrings might overlap each other. That is, instead of beginning the second substring one bit position beyond the end of the first substring, we might begin the second substring at the second bit position in the first substring, thereby overlapping the two substrings by five bits. Thus, five of the six bits in the first substring also contribute their part to the fitness contribution of the second substring, and so on, for all  $N$  bit positions. When this is the case, there are a total of 60 substrings of length 6 (and  $K = 5$ ) in a string where  $N = 60$ . We allow  $K$  of the substrings to wrap around from the end of the main string to its beginning whenever the start position of the substring is greater than position  $N - K + 1$  in the main string. The fitness contribution of each of the  $N$  substrings is dependent on the  $2^{K+1}$  possible position combinations of the substring's bits. Thus, there are  $N$  tables, each with  $2^{K+1}$  function values that have been randomly chosen with a uniform distribution over the interval,  $[0,1)$ . These function values additively contribute to the

overall fitness of the string. (See Figure 3.3 and Figure 3.4 and accompanying example on the next page.) The two main parameters in the *NK* model are the number of bits in the string and the number of other bits that epistatically influence the fitness contribution of each substring. In the *NK* landscapes, there is a high degree of coupling between bits that are close to each other in the string. *NK* functions are non-separable. Simultaneously mutating bits that have a high degree of epistatic interaction is one of the areas this work investigates.

For this study, a fresh instance of a randomly generated *NK* function is produced for each set of hillclimbs performed. This is because randomness is part of the definition of the *NK* landscapes. This is unlike the runs using the easy and deceptive landscapes. For those, the same function is used for all hillclimbs.

**Figure 3.3.** Elements of an  $NK$  function where  $N = 6$  and  $K = 2$  (substring length = 3).

	locus[0]	locus[1]	locus[2]	locus[3]	locus[4]	locus[5]
000	0.25	0.55	0.19	0.23	0.23	0.77
001	0.20	0.02	0.99	0.38	0.22	0.08
010	0.85	0.66	0.03	0.71	0.09	0.17
011	0.52	0.01	0.71	0.16	0.55	0.42
100	0.01	0.77	0.13	0.83	0.76	0.98
101	0.22	0.22	0.30	0.53	0.69	0.44
110	0.06	0.61	0.55	0.53	0.16	0.49
111	0.51	0.88	0.00	0.50	0.66	0.38

Locus	Neighborhood	Fitness
0	010	0.85
1	101	0.22
2	011	0.71
3	110	0.53
4	100	0.76
5	001	0.08

**Figure 3.4.**  $NK$  Fitness calculation for the string *101100*. The contribution of locus[0] is influenced by bits in positions 5 and 1. The contribution of locus[1] is influenced by positions 0 and 2, and so on.

$$\text{Fitness} = (0.85 + 0.22 + 0.71 + 0.53 + 0.76 + 0.08) / 6 = (3.15 / 6) = 0.525$$

### 3.4 Formal Statement of Hypotheses and Observations

- We hypothesize that the macromutation algorithms, *MMHC1* and *MMHC2* perform better than *RMHC* on all test problem sizes. This is demonstrated emphatically on the non-separable, *NK* landscapes.
- We hypothesize that mutating bits that are likely to be affine in the fitness function (*MMHC1*) results in somewhat lower performance levels on all test problems than mutating bits that are likely to be strongly coupled in the fitness function (*MMHC2*).
- We hypothesize that for all problem sizes, *RMHC* performs better on separable functions, the easy and deceptive problems, than it does on the non-separable *NK* landscapes.
- We observe the effect on algorithm performance of changes in the *emoves* and *nmoves* parameters.
- We observe the effect of the problem length,  $N$ , on algorithm performance.



# Chapter 4

## The Experiments

In order to produce reliable results from this investigation, it was necessary to conduct a sufficient number of trials for each of the possible settings of the various algorithm parameters on each of the proposed problem sizes. All of the trials then had to be repeated using each of the three different algorithms. Initially, decisions had to be made regarding the range through which each parameter would vary.

### 4.1 Parameters

#### 4.1.1 $N$

Specifically, for all algorithms tested, we wanted to see if the size of the problems, the bit length  $N$ , affected algorithm performance. Since previous work by Jones[3] with hillclimbing algorithms had been done using problem sizes generally less than 120 bits, we chose to start with string lengths of 60 bits and 120 bits, but also to extend this to 180 bits.

### 4.1.2 *n*moves and *e*moves

As well, we wished to observe the effects of varying the parameters *n*moves and *e*moves in all cases, since these parameters are crucial in determining how thorough an algorithm's search of a landscape for highly fit points will be. Again relying on previous work for guidance as to the range of parameter values selected, we choose to tie them to the value of  $N$  in each case. We use  $N/2$ ,  $2N$  and  $3N$  for the settings of *n*moves and  $N/2$ ,  $10N$  and  $50N$  as the settings for the *e*moves parameter. Reasoning here was that tying the value of these parameters to  $N$  would result in their being more fairly comparable across all algorithm runs and problem sizes.

### 4.1.3 *bits*

The number of bits mutated is a very important parameter to observe. Of course, in the *RMHC* algorithm, *bits* is always, trivially, 1. With *MMHC1* and *MMHC2* however, we vary *bits*. We choose to vary it between 2 and 6. Six is the number of bits in each of the fully easy and fully deceptive subproblems used in our study. Thus it might be possible, in a single iteration of an algorithm, to simultaneously mutate all of the bits in one of the easy or deceptive subproblems. On the  $NK$  landscapes, we had chosen the largest substring size to be six. Therefore, on those subproblems it also might be possible, in one iteration, to simultaneously mutate all of the bits in one substring.

#### 4.1.4 $K$

One way to view  $K$  is that it refers to the length of a substring on the  $NK$  landscapes. (Kauffman's definition of  $K$  in the  $NK$  landscapes is one less than the number of bits in the length of a substring. The parameter  $K$  that we use in this study is equal to Kauffman's  $K$ , plus one.) The value  $K = 6$  is also the subproblem size we use for the fully easy and fully deceptive landscapes. With the latter two landscapes, we choose a fixed subproblem size of six since we use the easy and deceptive functions designed by Goldberg[7] and used in Jones's work[3]. With the  $NK$  landscapes however, we were able to vary the subproblem size. On the  $NK$  landscapes, we vary the subproblem size between 2 and 6. Letting the parameter  $K$  be greater than 6 on the  $NK$  landscapes results in function tables that quickly become unwieldy, due to the fact that each  $NK$  function contains  $N$  tables of  $2^K$  randomly generated values.

**Figure 4.1.** Summary of elements comprising the hillclimbs.

Algorithms	<i>RMHC</i>	<i>MMHC1</i>		<i>MMHC2</i>		
Landscapes	Fully Easy	Fully Deceptive		$NK$		
$N$	60	120	180			
$K$	2	3	4	5	6	
<i>bits</i>	1	2	3	4	5	6
<i>nmoves</i>	$N/2$	$2N$	$3N$			
<i>emoves</i>	$N/2$	$10N$	$50N$			

## 4.2 Procedures

### 4.2.1 Implementation

The *RMHC*, *MMHC1* and *MMHC2* algorithms were implemented in the *C* language using a suite of *C++* classes developed by Alden H. Wright at The University of Montana in 1996. In particular, from this suite, the class that provides a bit sequence and its accompanying functionality was used, as was the class that provides for the generation of random numbers. All algorithms and accompanying functions were written in *C*.

### 4.2.2 Preliminary Hillclimbs

Once the parameter range settings were decided, and the code to implement the algorithms written, a preliminary set of hillclimbs was performed. The objective of performing this preliminary set of hillclimbs is to determine the amount of work that we require each of the three algorithms to accomplish in order to compare them fairly. We decided to use as our unit of measure a single evaluation of string fitness. The total number of times that the fitness function is called on to evaluate string fitness during ten complete hillclimbs, using each of the parameter setting combinations, represents the amount of work the algorithm does on a given combination of parameter settings. The test problems are sufficiently hard that there is a difference in performance among the algorithms. We chose to allow each algorithm/setting combination ten hillclimbs. We considered that this would present sufficient opportunity for the algorithms to return a measure of work done that reflects differences among them. (See Figure A.2 in the Appendix for a sample command-line call to the *MMHC2* algorithm for the preliminary hillclimbs.) We retain across all runs the largest number of function evaluations required

for each set of ten hillclimbs. The resulting single numeric measure is later used as the standard on which to compare the performance of the different algorithms. On the 60-bit problems, especially with the lower settings of the *nmoves* and *emoves* parameters, the number of function evaluations required for an algorithm to complete ten hillclimbs can be relatively small compared with the number required for the 180-bit problems using the higher settings of *nmoves* and *emoves*.

After all preliminary hillclimbs were run, we found that the largest number of function evaluations required for any algorithm to finish ten hillclimbs was nearly 5.7 million. This number is the yardstick for our final comparisons. The single algorithm whose maximally fit point is the best after each algorithm performs 5.7 million function evaluations can be considered the better-performing algorithm, using a given combination of parameter settings.

### 4.2.3 Final Hillclimbs

Fairly comparing all the algorithms using every combination of the parameter settings meant performing however many hillclimbs it took, using each setting, to obtain 5.7 million function evaluations. (See Figure A.3 in the Appendix for a sample command-line call to the *MMHC2* algorithm for the final hillclimbs.) From the 5.7 million evaluations allowed for each different combination of the settings, we retained the maximum fitness value that was achieved in all the resulting hillclimbs. That value, and the parameter settings that produce it, were output to a file used to collect the data. As soon as an algorithm would complete 5.7 million evaluations, it outputs its data to the

data file, proceeds to the next combination of parameter settings and begins another 5.7 million evaluations. Three UNIX C-Shell scripts were used to run the algorithms through all of the necessary parameter combinations. (See Figure A.1 in the Appendix for an example UNIX shell script for running *MMHC2*.) For each of the three algorithms, this process was repeated for each combination of parameter settings until all combinations had been used. In the entire work, there are a total of 2,079 separate combinations of parameter settings observed. (See Figure 4.1.) That total results from 945 setting combinations being used in each of the *MMHC1* and *MMHC2* algorithms and 189 setting combinations tried for *RMHC*. Not absolutely every parameter setting was combined with absolutely every other one on all three algorithms, however. For instance, it must be remembered that since *RMHC* is the algorithm that mutates only one bit at a time, the *bits* parameter for *RMHC* is always set at 1. Similarly, the *MMHC1* and *MMHC2* algorithms do not use a setting of 1 for the *bits* parameter, since data for settings which used *bits* = 1 were obtained from runs of *RMHC*. (Additionally, it makes no sense to say that we macromutationally vary one bit.) Also, the value of *K* was set at a constant 6 for all algorithms on the fully easy and fully deceptive landscapes. This is because 6 bits is the size of the subproblems used in the test landscapes.

Each combination of parameter settings required 5.7 million function evaluations. This resulted in nearly 12 billion function evaluations being observed during the entire study. Completion of these final runs took several weeks of computation time, even with the algorithms simultaneously being run on many different computers.

# Chapter 5

## Data and Results

Upon completion of all algorithm runs, the data files produced by those runs were used as input to spreadsheet software for conversion to spreadsheets and subsequently for conversion to charts used in analyzing the data and producing results. A discussion of the conclusions of this investigation, referencing these charts, is in Chapter 6.

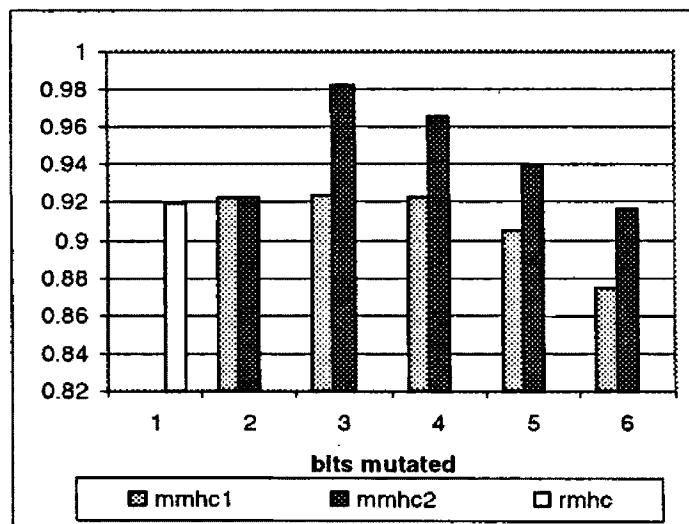
### 5.1 All Algorithms / All Problem Sizes

#### 5.1.1 All Algorithms: bits mutated

The chart shown in Figure 5.1 is a general overview of the entire study using the fully deceptive landscape. It clearly shows the superior results obtained with the *MMHC2* algorithm across all tested problem sizes.

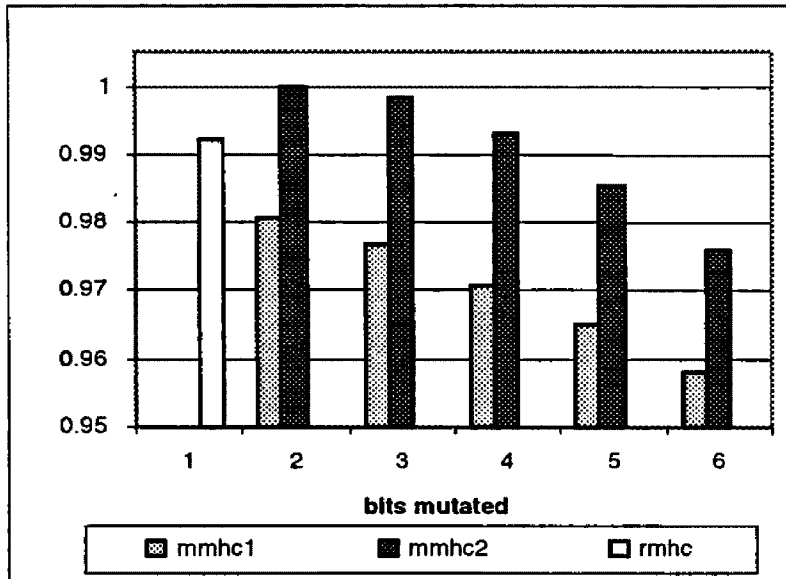
##### 5.1.1.1 Deceptive

**Figure 5.1.** For a *bits* setting of 3, 4 or 5, *MMHC2* shows remarkably good performance on the fully deceptive landscape. (A *bits* setting of 1 is valid only for *RMHC*.) For the macromutation algorithms, generally, as *bits* increases past 2 or 3, average fitness declines.

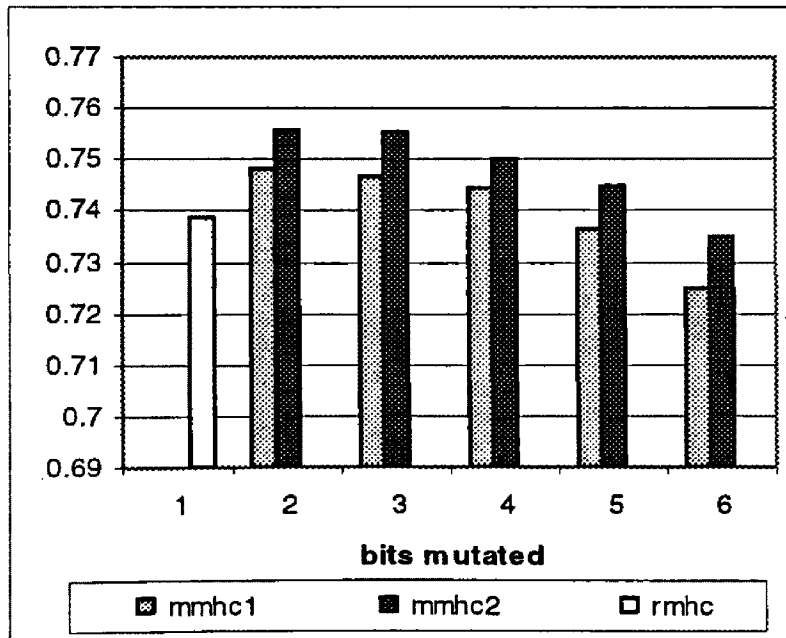


### 5.1.1.2 Easy and *NK*

Similar charts are produced for the fully easy landscape and the *NK* landscapes. On the easy landscape *MMHC2* is clearly the better performer, when the number of mutated bits is set to 2, 3 or 4. On the *NK* problems, *MMHC2* is better than *MMHC1* or *RMHC* using a *bits* setting between 2 and 5.



**Figure 5.2.** We see the average fitness achieved with all the algorithms on the fully easy landscape. Again, we see outstanding results obtained with *MMHC2*, especially for a *bits* setting of 2 or 3. Notable are the relatively low fitness values for *MMHC1* and the fact that *RMHC* gets better numbers than *MMHC1* throughout. *RMHC* even beats *MMHC2* with *bits* = 6.



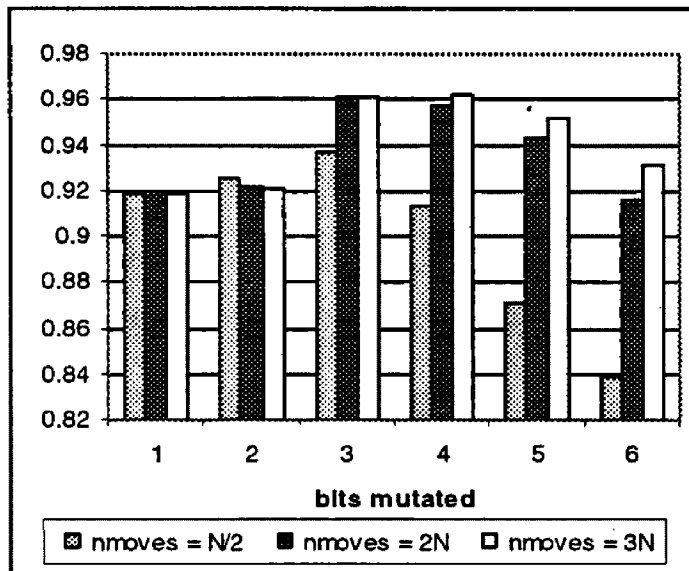
**Figure 5.3.** As expected, the average fitness reached, for all algorithms on the *NK* landscapes is somewhat better for the macromutation algorithm, *MMHC2*, than for *RMHC* for most *bits* settings. However, on the *NK* landscapes, *MMHC1* doesn't lag as far behind *MMHC2* as it does on the previous two charts.



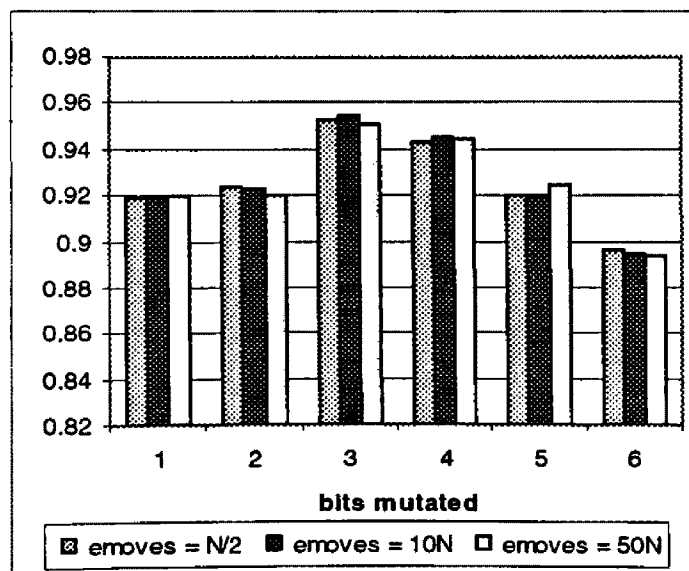
### 5.1.2 All Algorithms: bits mutated / *n*moves and *e*moves

The charts showing the effect of varying the *n*moves and *e*moves parameters across all algorithms and problem sizes studied generally tend to support our belief that adjusting these factors would make a difference in average fitness values achieved. What is surprising is that various settings of the *n*moves parameter show greater differences in average fitness than adjustment of the *e*moves parameter.

#### 5.1.2.1 Deceptive

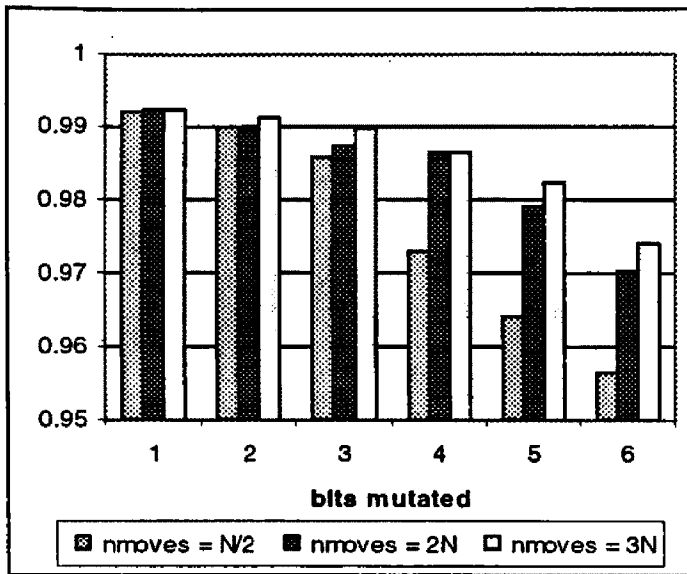


**Figure 5.4.** Changes in *n*moves on average fitness for the deceptive problems are relatively insignificant for the smallest number of bits mutated. However, as *bits* increases, it is clear that the *n*moves parameter is vital to improved average fitness numbers, as shown by the rapid decline for  $N/2$ .

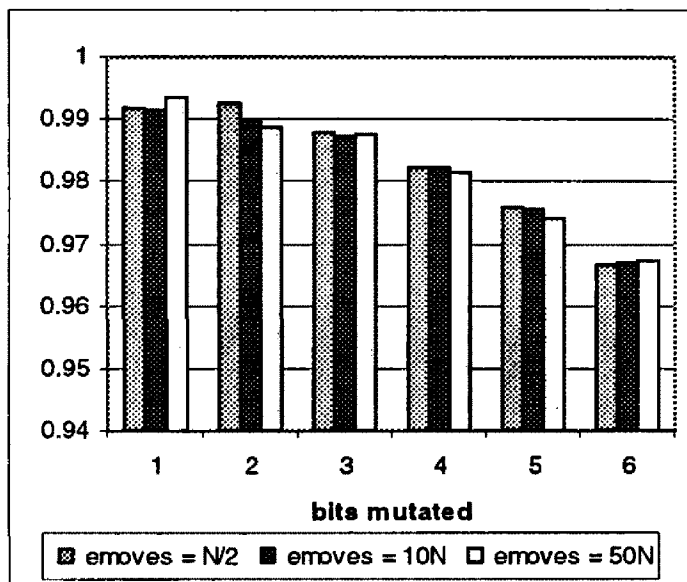


**Figure 5.5.** For deceptive problems, the *e*moves parameter does not exhibit effects that are as profound as with *n*moves. In fact, an increase in *e*moves beyond  $10N$  tends to give decreased average fitness for most settings of *bits*. It only results in increased fitness at *bits* = 3 and *bits* = 4.

### 5.1.2.2 Easy

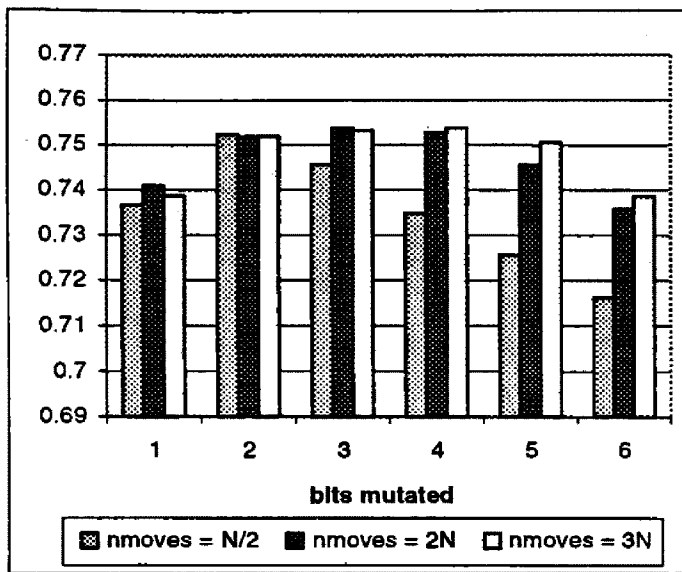


**Figure 5.6.** On the fully easy landscapes, *nmoves* exhibits similar tendencies as it does on the deceptive problems; an increase in *nmoves* becomes more critical to greater average fitness numbers as the *bits* parameter gets bigger.

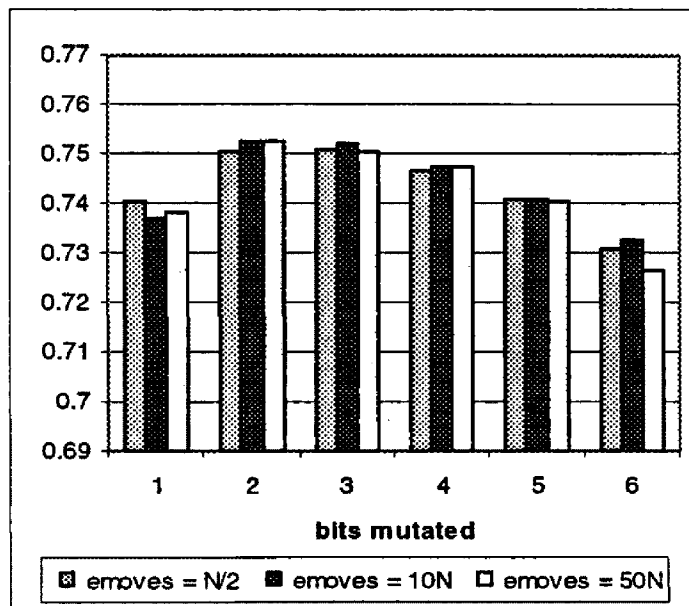


**Figure 5.7.** While the *emoves* parameter appears to contribute some to better average fitness values, again with the easy problems, a variation of *emoves* seems to effect relatively insignificant changes in average fitness at each of the *bits* settings.

### 5.1.2.3 NK Landscapes



**Figure 5.8.** On the *NK* landscapes, as with the previous problems, changes to the *nmoves* parameter seem to have more effect on average fitness as the size of the *bits* parameter varies from 2.

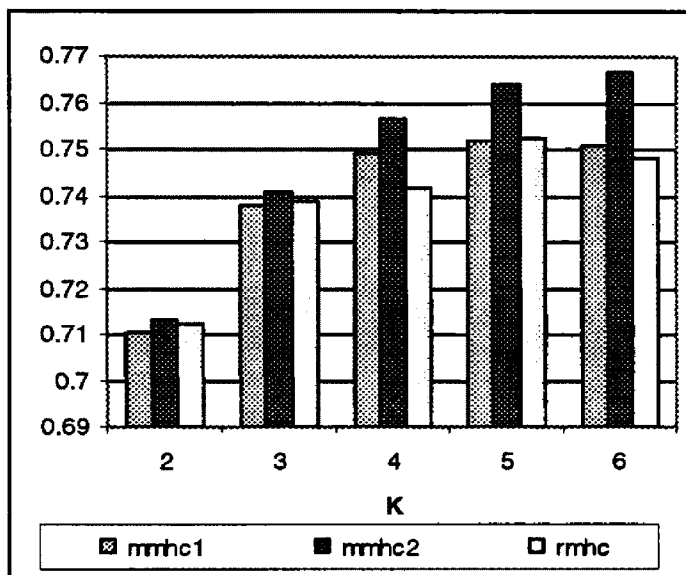


**Figure 5.9.** The *emoves* parameter on the *NK* landscapes causes a slightly different trend in average fitness, compared with the previous two problems. As *bits* increases, an *emoves* setting of  $10N$  seems to emerge as the better setting despite the fact that overall, average fitness tends to decline with an increase in *bits* beyond *bits* = 2.

### 5.1.3 All Algorithms: $K$

The chart shown in Figure 5.10 is a general overview of the entire study for different values of the parameter  $K$ , on the  $NK$  landscapes. Since we use a fixed value of  $K = 6$  on the deceptive and easy problems, we have no data for variations of  $K$  on those landscapes. Since we have no data on the global maximum for each  $NK$  landscape, the  $NK$  data are not as definitive as the data for the easy and deceptive problems.

#### 5.1.3.1 $NK$ Landscapes

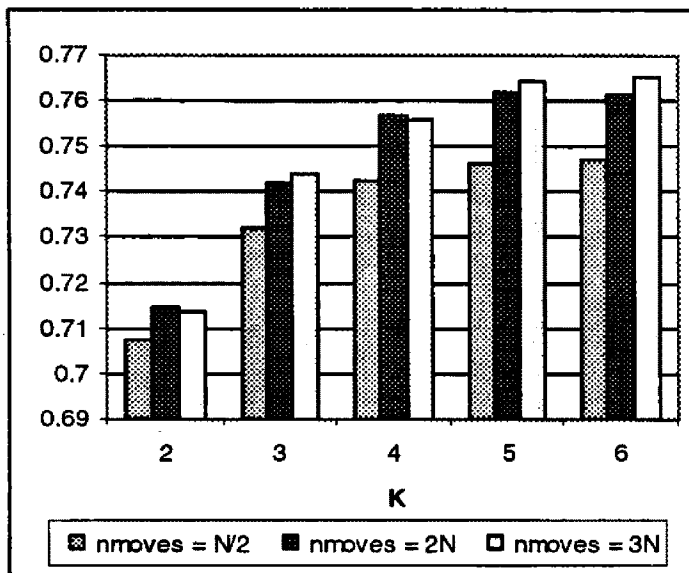


**Figure 5.10.** This chart also shows the good average fitness values achieved with *MMHC2* on the  $NK$  landscapes. In general, as the parameter  $K$  is increased in the range 2 - 6, the fitness values for *MMHC2* improve. The average fitness for the other algorithms improves also, but only until  $K = 5$ , after which their numbers decline.

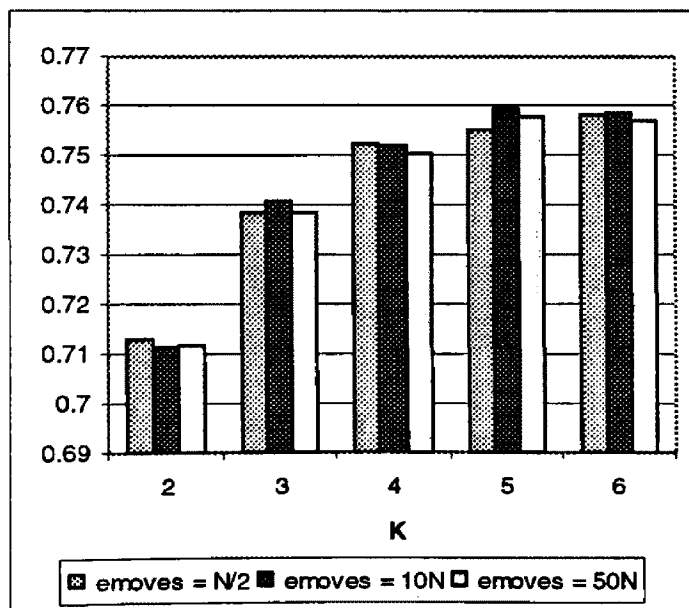
### 5.1.4 All Algorithms: $K / nmoves$ and $emoves$

We see the effect of varying the  $nmoves$  and  $emoves$  for different  $K$ . Again, only results for the  $NK$  landscapes are shown since we used a fixed  $K = 6$  on the easy and deceptive landscapes. An important result is that in general, the average fitness achieved by all algorithms on combined problem sizes seems to improve as the parameter  $K$  increases.

#### 5.1.4.1 $NK$ Landscapes



**Figure 5.11.** For all settings of the  $K$  parameter, the two higher settings of  $nmoves$  show higher average fitness values. As  $K$  increases, so does the dominance of the higher two values of  $nmoves$ , with the  $3N$  setting being the better of the two in most cases.

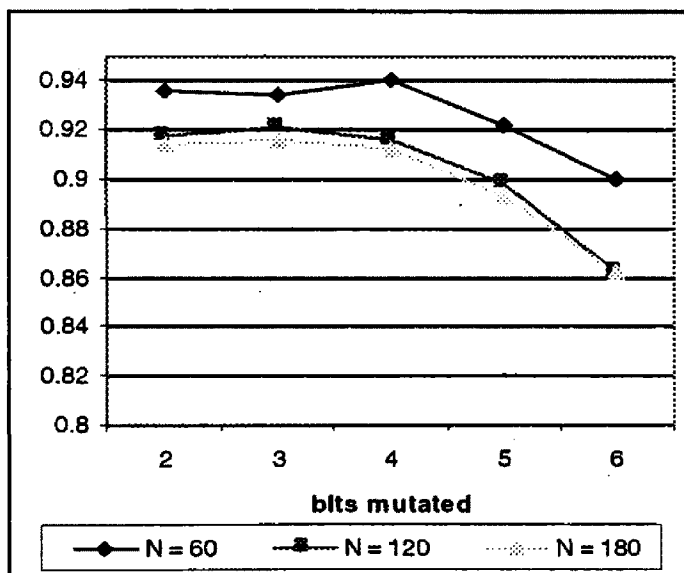


**Figure 5.12.** Somewhat less dramatic differences in average fitness are noted with variations in the  $emoves$  parameter than with the  $nmoves$  parameter. Here, the effect of different settings of  $emoves$  tends to be a bit more pronounced for  $K = 5$ , at which point  $emoves = 10N$  does best.

## 5.2 *MMHC1* / All Problem Sizes

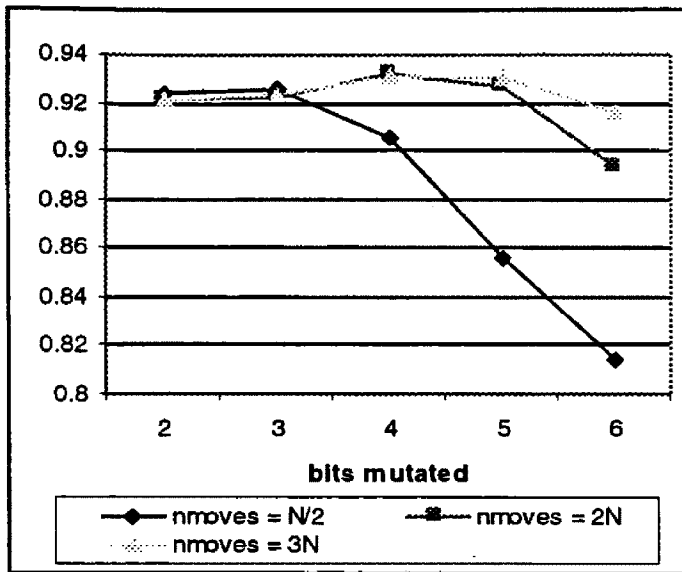
On the following pages are charts depicting data for the different landscapes in which all problem sizes are combined. There is no data for variations in  $K$  on the deceptive and easy landscapes, since  $K$  is constant 6 on them. However, varying data for  $K$  are shown on charts for the  $NK$  landscapes. We begin with data for the distributed algorithm, *MMHC1*.

### 5.2.1 *MMHC1* / Deceptive: bits mutated

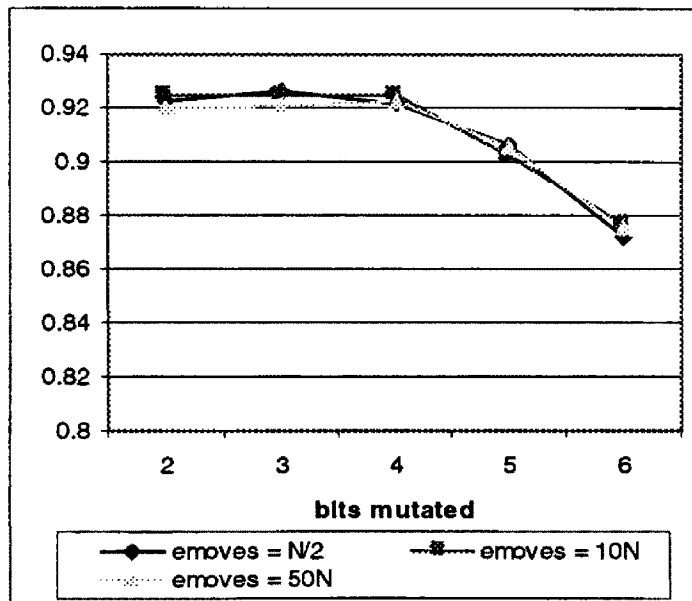


**Figure 5.13.** As the *bits* parameter becomes larger, beyond a value of 4, the average fitness values for the *MMHC1* algorithm on the deceptive problems fall off sharply. This is true for all tested problem sizes. It is interesting to note that the algorithm does better on this landscape overall on the 60 bit problems and somewhat less well on the 120 and 180 bit problems.

### 5.2.2 MMHCI / Deceptive: bits mutated / *n*moves and *e*moves

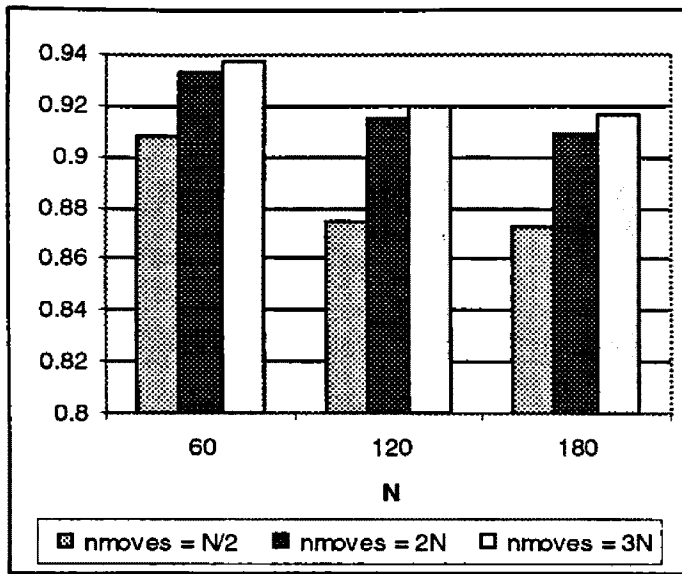


**Figure 5.14.** The same overall trend of decreased average fitness values with an increase in the *bits* parameter is seen here for MMHCI. The *n*moves parameter makes a dramatic difference in average fitness for the deceptive landscape, especially when *bits* is greater than 2.

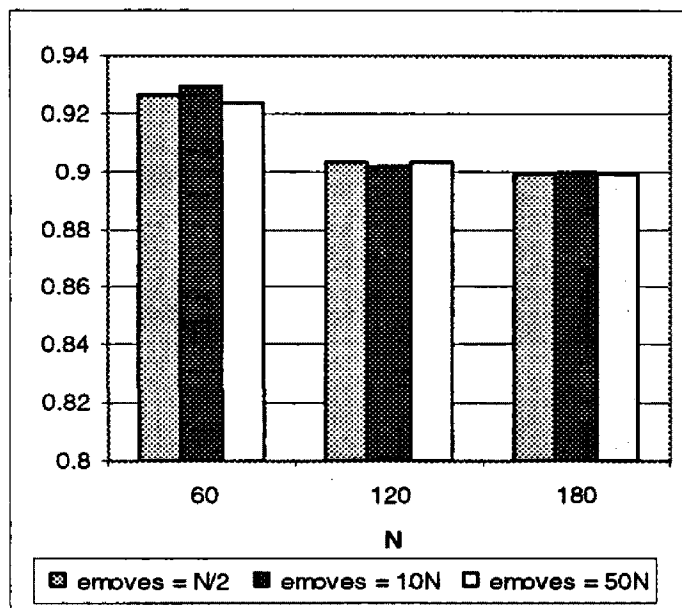


**Figure 5.15.** Adjusting the *e*moves parameter appears to have little effect on MMHCI on the deceptive landscape, for all *bits* values.

### 5.2.3 MMHC1 / Deceptive: $N$ / $n$ moves and $e$ moves



**Figure 5.16.** It can be seen here that on the deceptive problems of all tested sizes, an  $n$ moves setting of  $3N$  is clearly the best one to use with *MMHC1*. The data for the shorter bit-length problems show better results than that for larger  $N$ .

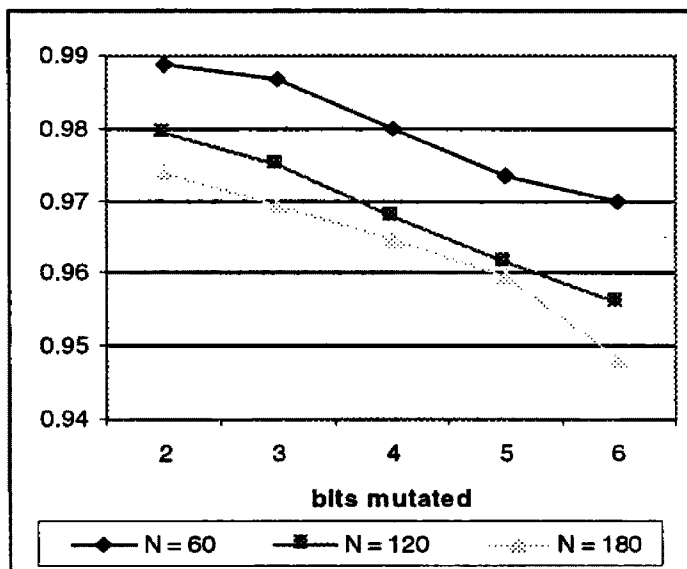


**Figure 5.17.** The data for the  $e$ moves parameter aren't quite as definitive as for  $n$ moves. It seems that changes to  $e$ moves produce very little effect on average fitness with deceptive problems of these sizes.



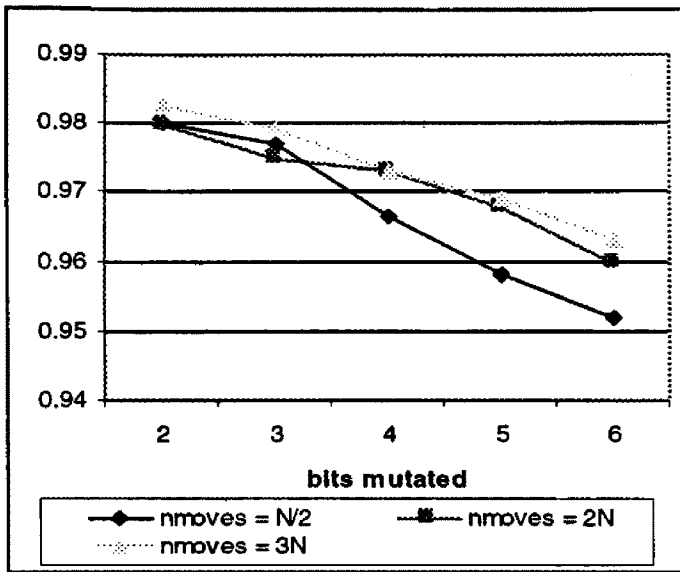
### 5.2.4 *MMHC1* / Easy: bits mutated

Results for *MMHC1* on the fully easy problems are in many ways similar to those for the deceptive problems. One major difference is that overall average fitness is significantly better for all algorithms on this landscape. Also, increasing the *nmoves* parameter has a less dramatic effect here.

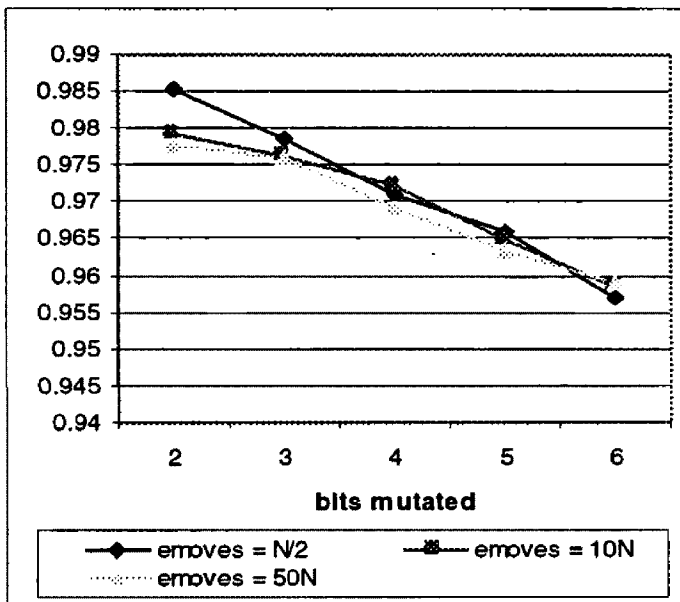


**Figure 5.18.** As *bits* is increased beyond 2, overall average fitness decreases, but not quite as much as on the deceptive landscapes. Again, the *MMHC1* algorithm returns higher average fitness values on the 60 bit problems, with a *bits* setting of 2.

### 5.2.5 MMHC1 / Easy: bits mutated / *n*moves and *emoves*

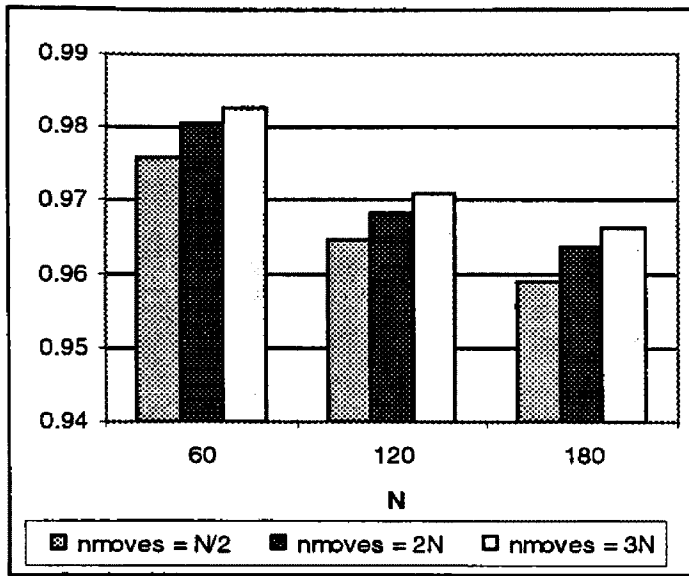


**Figure 5.19.** The  $3N$  setting of *n*moves for MMHC1 on all the easy problem sizes tested seems to be the better setting. The *bits* setting of 2 seems to be best here, as well.

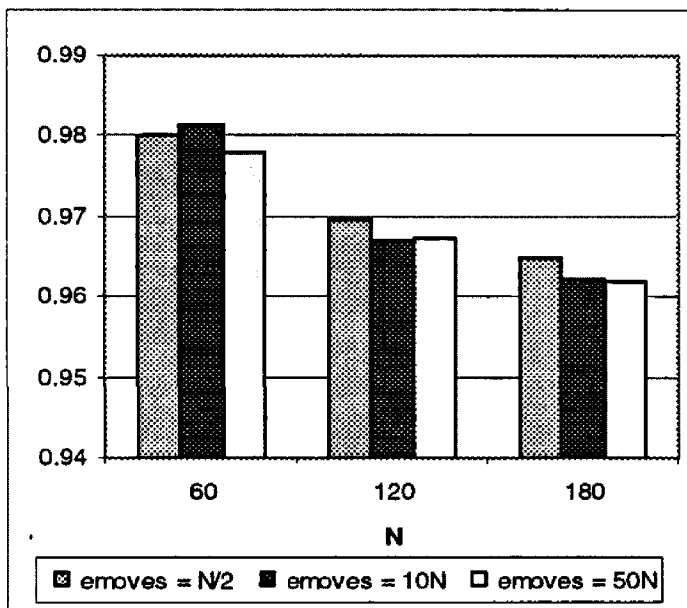


**Figure 5.20.** Variations in the *emoves* parameter appear to produce rather insignificant changes in the average fitness at all *bits* settings, as *bits* increases beyond 2.

### 5.2.6 MMHCI / Easy: $N$ / $n$ moves and $emoves$



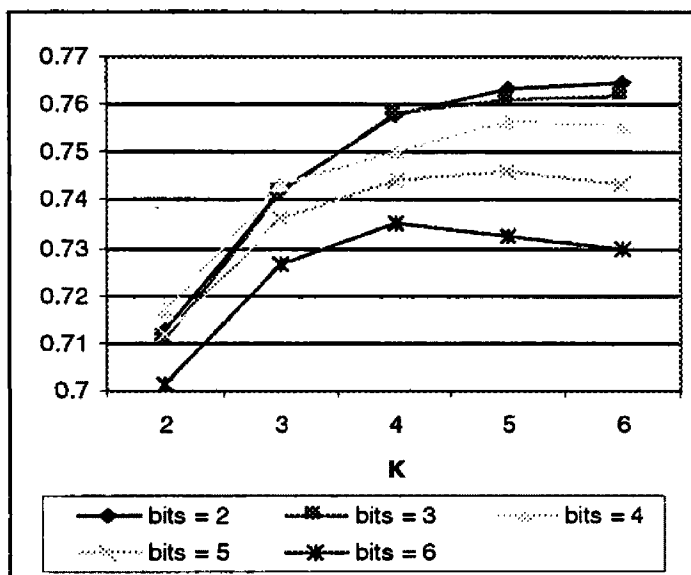
**Figure 5.21.** The *MMHCI* *n*moves setting of  $3N$  again is the best setting for all tested problem sizes on the easy landscapes.



**Figure 5.22.** The lower two settings for the *emoves* parameter turn out to give slightly better results on most easy problem sizes here.

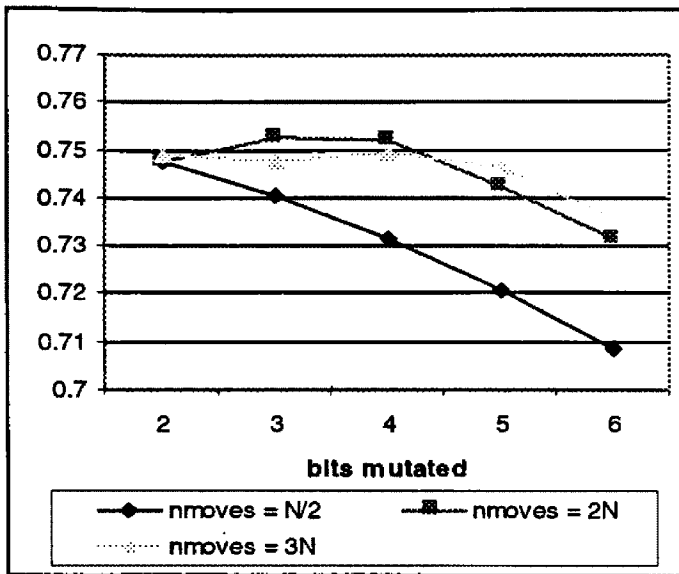
### 5.2.7 *MMHC1* / *NK* Landscapes: bits mutated / $K$

Since we vary  $K$  on the *NK* landscapes, the data presented for the *NK* problems is more complex than for the easy and deceptive landscapes. Since we do not know the global maximum for each *NK* landscape, average fitness data for our three algorithms on the *NK* landscapes are not as definitive as they are with the separable functions.

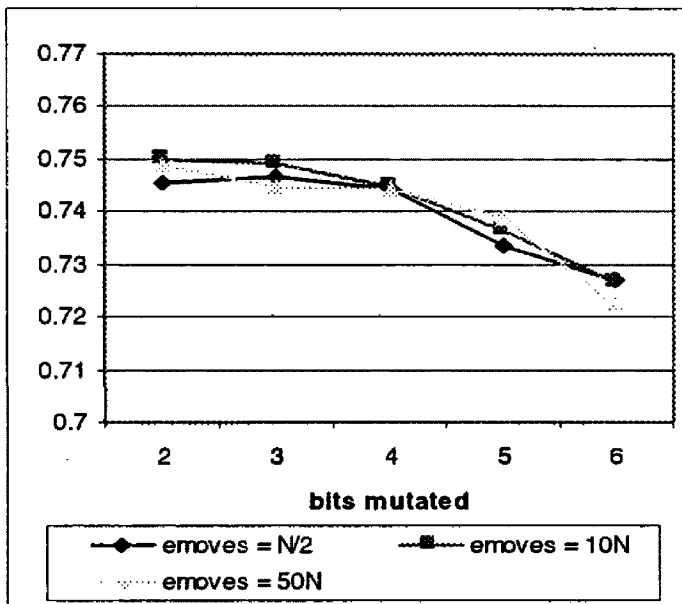


**Figure 5.23** On the *NK* landscapes, we see the effect of varying the  $K$  parameter. The average fitness values for *MMHC1* are seen here to show greater variation above the value of  $K = 4$ , for *bits* settings between 2 and 6. The larger settings of *bits* produce much poorer results, with average fitness declining above  $K = 4$ .

### 5.2.8 MMHC1 / NK Landscapes: bits mutated / *n*moves and *e*moves

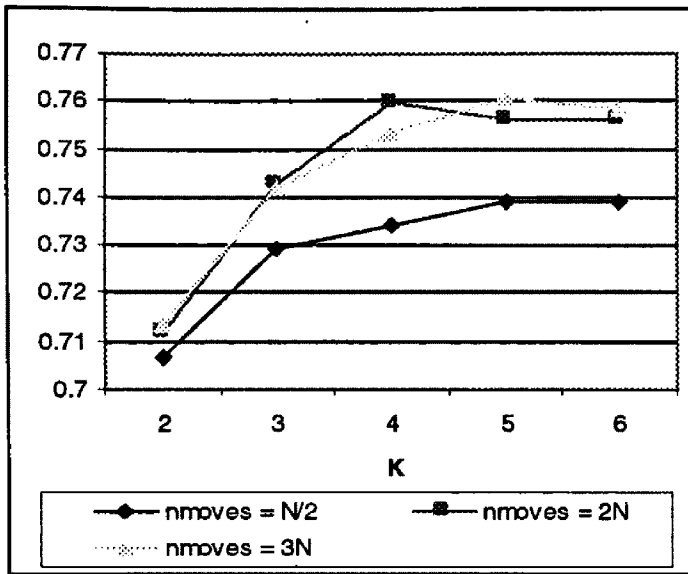


**Figure 5.24.** The average fitness reached by MMHC1 on the NK landscapes appears to be highest when *bits* = 3 or *bits* = 4 and we let *n*moves = 2*N*. Increasing *bits* beyond 4 causes a rapid decline in fitness for all three values of *n*moves.

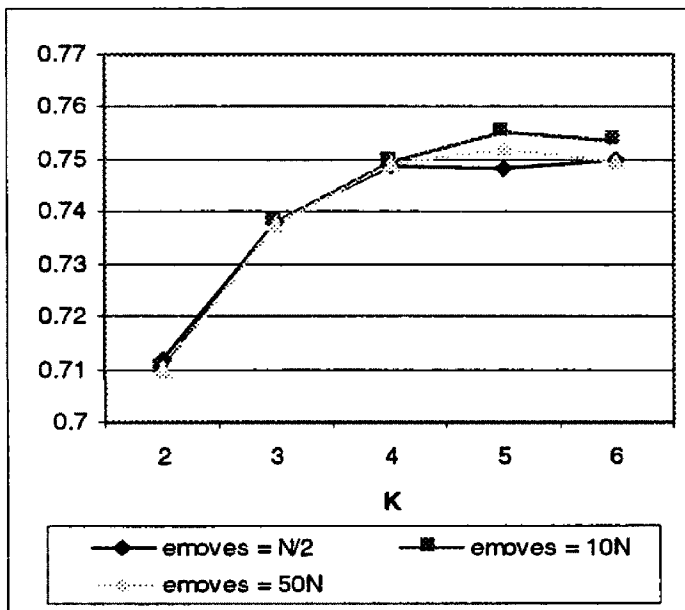


**Figure 5.25.** Increasing the *bits* parameter past 3 has its expected effect of a decrease in average fitness with MMHC1 on the NK problems, too. Varying *e*moves has almost no effect here, at all *bits* settings.

### 5.2.9 MMHC1 / NK Landscapes: $K$ / $n$ moves and $e$ moves



**Figure 5.26.** As the  $K$  parameter increases with MMHC1 on the NK landscapes, overall fitness also increases, up to  $K = 4$ . Here, the larger two values of the  $n$ moves parameter seem to contribute more to the average fitness numbers.

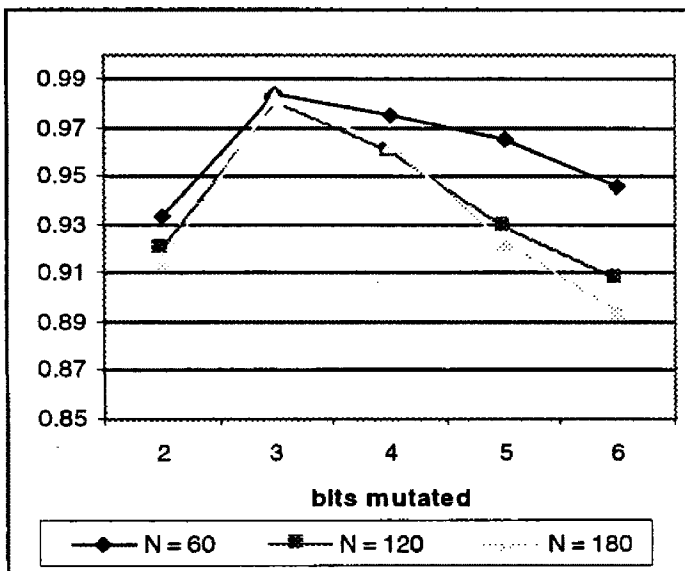


**Figure 5.27.** While the effect of varying the  $e$ moves parameter isn't quite as large as it is with the  $n$ moves parameter, as  $K$  increases past a setting of 4, an  $e$ moves setting of  $10N$  seems to do slightly better than the other settings.

### 5.3 MMHC2 / All Problem Sizes

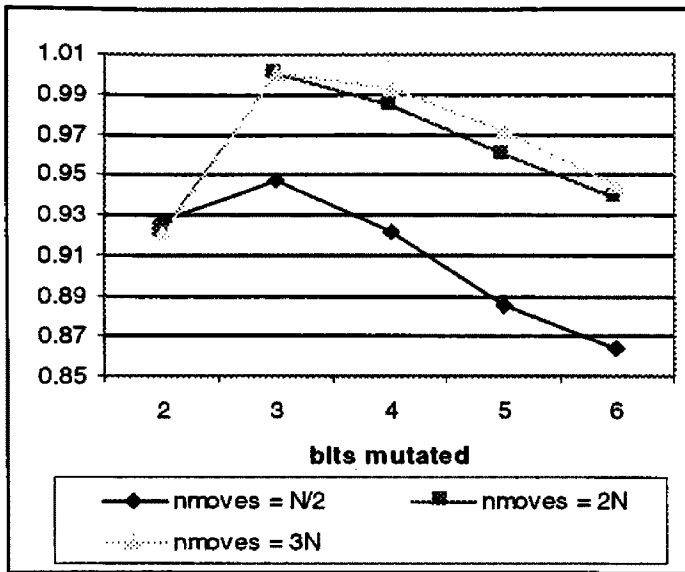
We present charts depicting the average fitness values achieved by the localized algorithm, *MMHC2*, on the next several pages. Data for each landscape are shown. In this section, the results are combined for all problem sizes we tested.

#### 5.3.1 MMHC2 / Deceptive: bits mutated

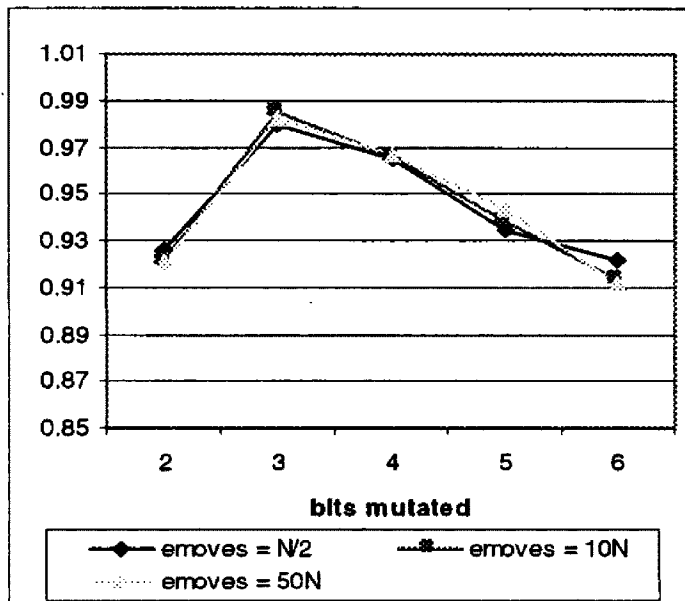


**Figure 5.28.** An interesting feature of the *MMHC2* data for the deceptive problems is the very dramatic peak in average fitness of this algorithm on all problem sizes at a *bits* setting of 3. This is followed by a comparatively gradual decline in average fitness past that point, with the algorithm doing slightly better on the 60 bit deceptive problems than it does on the others.

### 5.3.2 MMHC2 / Deceptive: bits mutated / *n*moves and *e*moves



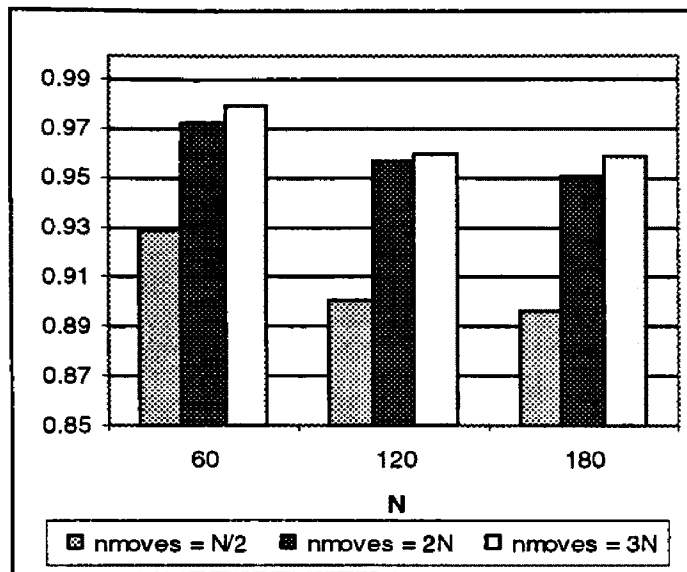
**Figure 5.29.** The dramatic peak in fitness at *bits* = 3 of the MMHC2 algorithm on the deceptive landscape is enhanced by the larger settings of the *n*moves parameter. Even at higher *bits* settings, when *n*moves =  $2N$  or *n*moves =  $3N$ , average fitness drops off more slowly. MMHC2 is even able to solve some of the deceptive problems, while using the higher two *n*moves settings



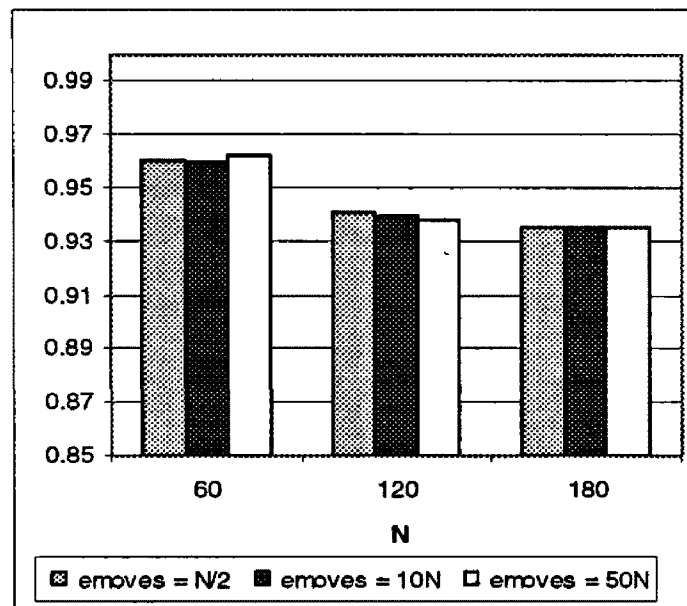
**Figure 5.30.** Once again, changes in the *e*moves parameter have a negligible effect on average fitness with the MMHC2 algorithm on these deceptive problems.



### 5.3.3 *MMHC2 / Deceptive: $N$ / $n$ moves and $e$ moves*



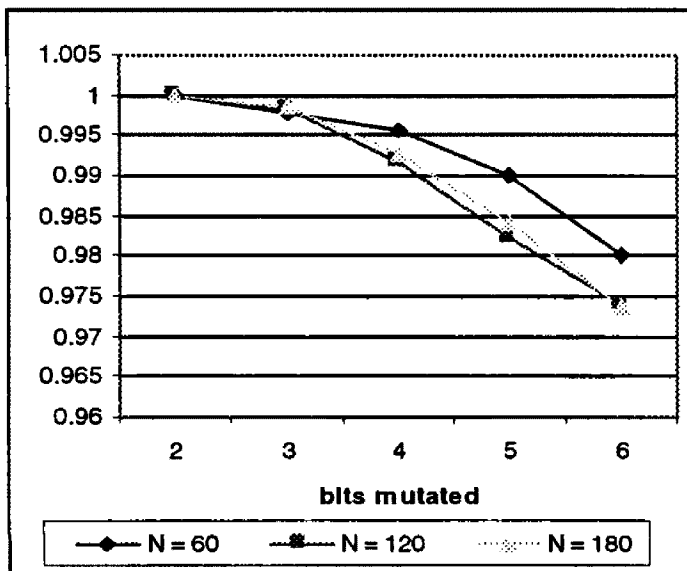
**Figure 5.31.** The smallest deceptive problems ( $N = 60$ ) do best with *MMHC2* in this chart, although the  $n$ moves parameter seems to contribute much to the average fitness of all three problem sizes. The larger  $n$ moves settings ( $2N$  and  $3N$ ) provide the best results.



**Figure 5.32.** The  $e$ moves parameter with the *MMHC2* algorithm on the deceptive problems appears to become even less important to average fitness as the problem size,  $N$ , increases.

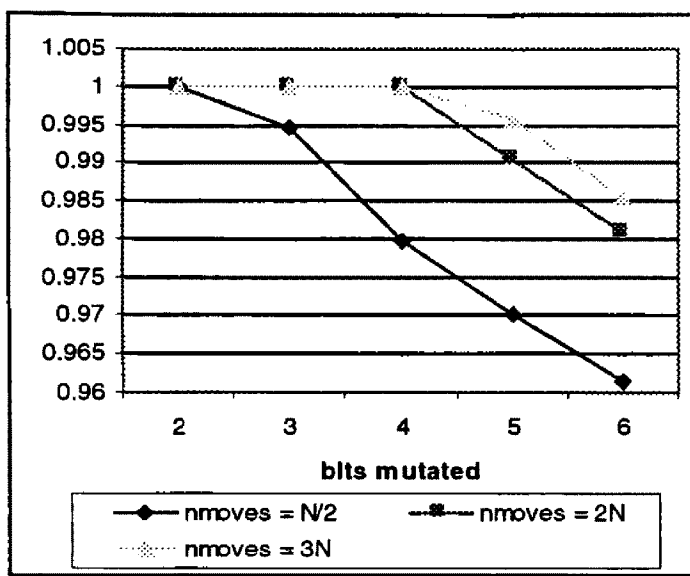
### 5.3.4 *MMHC2* / Easy: bits mutated

On the easy landscape, the *MMHC2* algorithm seems to solve the 6 bit subproblem quite easily (as the name implies) for all of the tested problem sizes, 60, 120 and 180 bits.

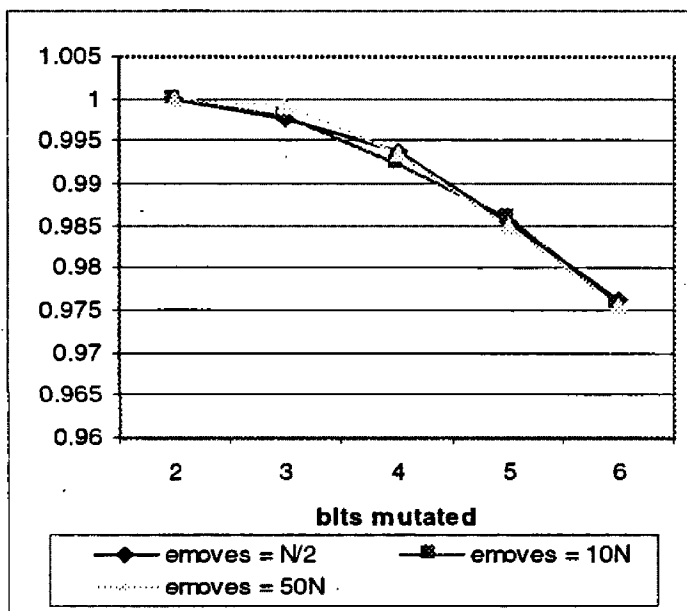


**Figure 5.33.** The 6 bit easy subproblems are solved by *MMHC2* for all problem sizes tested when *bits* = 2. Solution happens less often as the *bits* parameter increases. Again, the algorithm seems to do best overall on the 60 bit problem size, with *bits*  $\geq 3$ .

### 5.3.5 MMHC2 / Easy: bits mutated / *n*moves and *e*moves

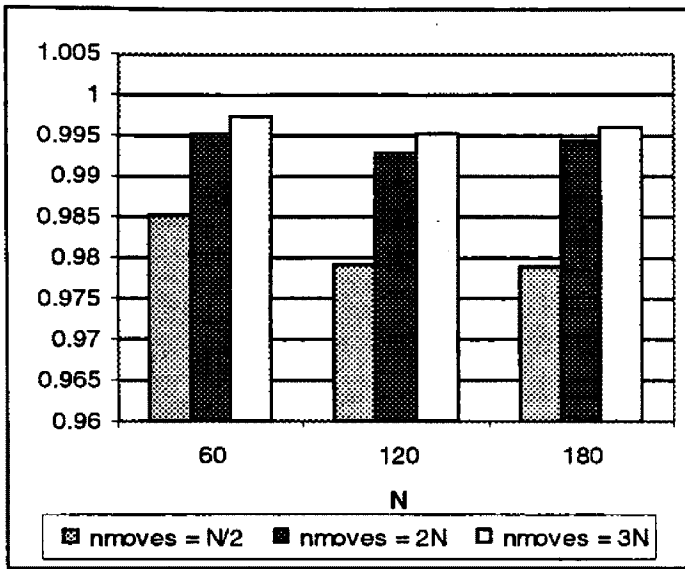


**Figure 5.34.** *MMHC2* is able to solve the fully easy problems of all tested sizes as long as the number of bits mutated is 4 or fewer and the *n*moves parameter is set at either  $2N$  or  $3N$ . The highest *n*moves setting,  $3N$ , seems to maintain somewhat better fitness numbers for *MMHC2* on these problems, above *bits* = 4.

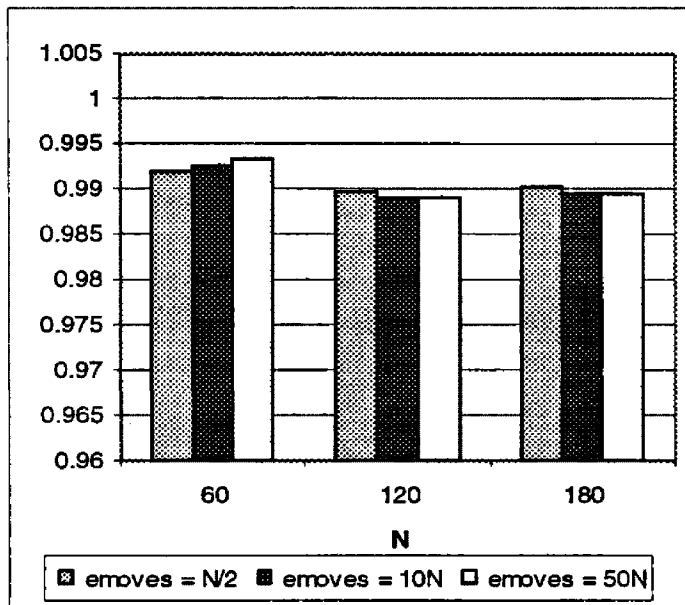


**Figure 5.35.** The *e*moves parameter does not seem to appreciably affect the overall average fitness the *MMHC2* algorithm is able to reach on this landscape. The trend of decreasing average fitness with an increase in *bits* is seen here as well.

### 5.3.6 MMHC2 / Easy: $N$ / $n$ moves and $emoves$



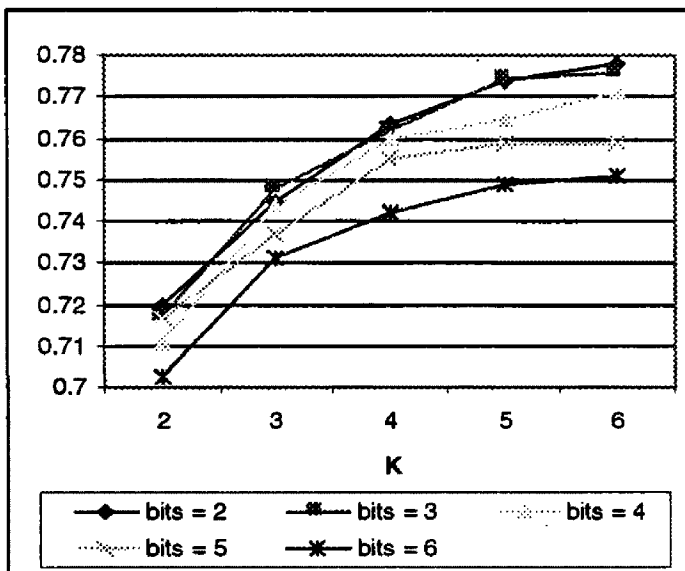
**Figure 5.36.** As we have seen in previous charts, the higher settings of the  $n$ moves parameter here also seem to greatly improve the average fitness of MMHC2 on all tested sizes of the easy problems.



**Figure 5.37.** Again, as we have seen earlier, the  $emoves$  parameter appears to have negligible effect on the ability of MMHC2 to do well on all tested sizes of the easy problems.

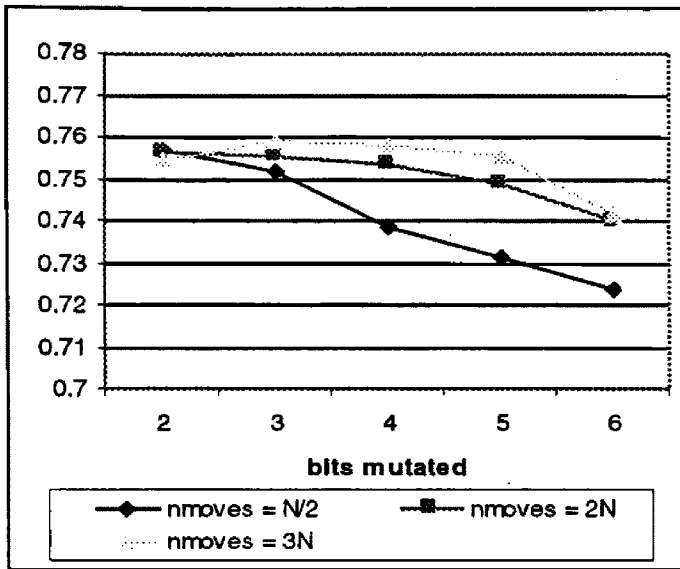
### 5.3.7 MMHC2 / NK Landscapes: bits mutated / $K$

Since we vary the  $K$  parameter only on the  $NK$  landscapes, these charts detail more complexity. The trends seen in the previous data continue here as well, with an increase in average fitness occurring as the  $K$  parameter increases, and average fitness decreasing as *bits* becomes greater past a certain point. Again, we remind the reader that since the global maximum of the  $NK$  landscapes is not known here, the  $NK$  results are not as definitive as they are with the separable easy and deceptive functions.

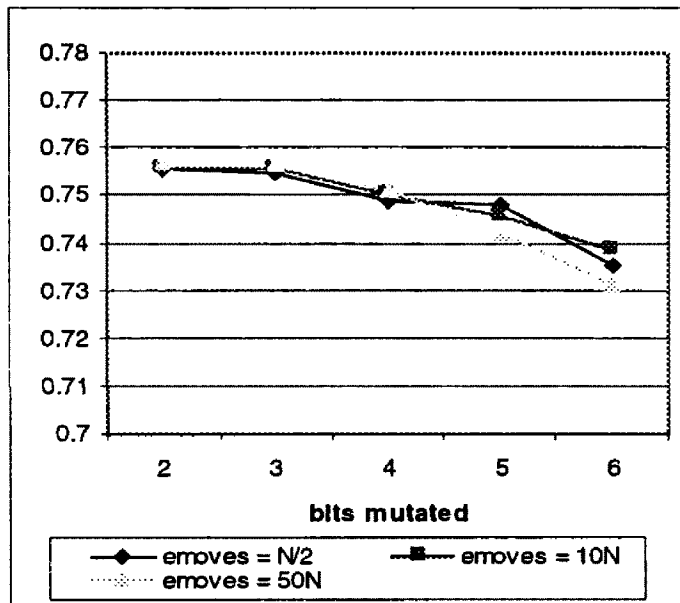


**Figure 5.38.** Not only does the average fitness achieved become greater with an increase in the  $K$  parameter, but increasing the *bits* parameter shows a significant decrease in average fitness. This is especially true for the higher settings of the  $K$  parameter.

### 5.3.8 MMHC2 / NK Landscapes: bits mutated / *n*moves and *e*moves

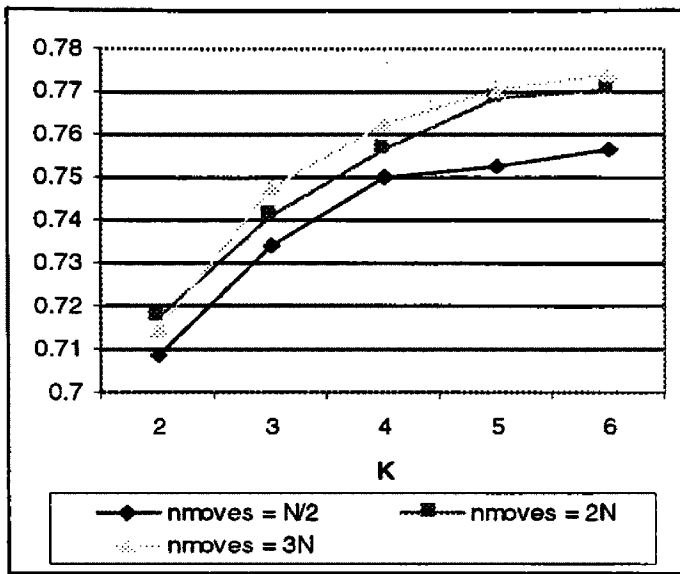


**Figure 5.39.** The *n*moves parameter seems to have the greatest influence on average fitness for MMHC2 on the NK landscapes. The most productive setting combination seems to be  $bits = 3$ ,  $n$ moves =  $3N$ .

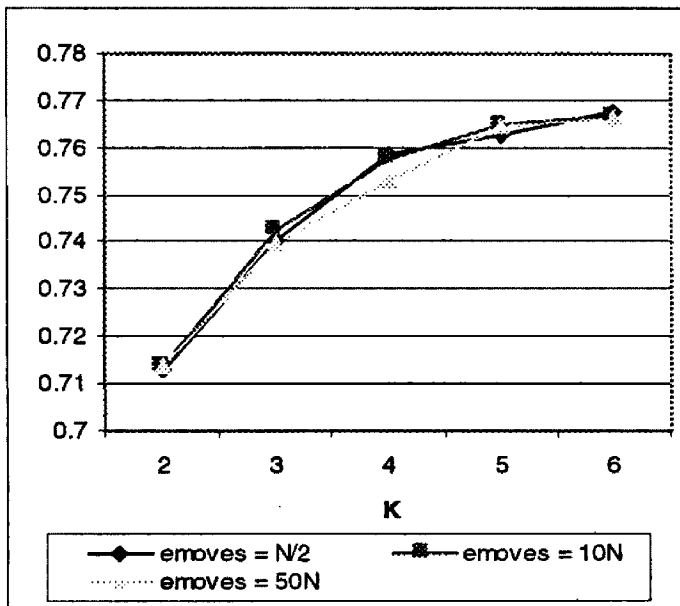


**Figure 5.40.** As *bits* is increased, average fitness of MMHC2 on the NK landscapes decreases. The *e*moves parameter does very little to affect that trend.

### 5.3.9 MMHC2 / NK Landscapes: $K$ / $n$ moves and $e$ moves



**Figure 5.41.** The larger values of the  $K$  parameter appear to produce better results here, especially when  $n$ moves =  $3N$ .

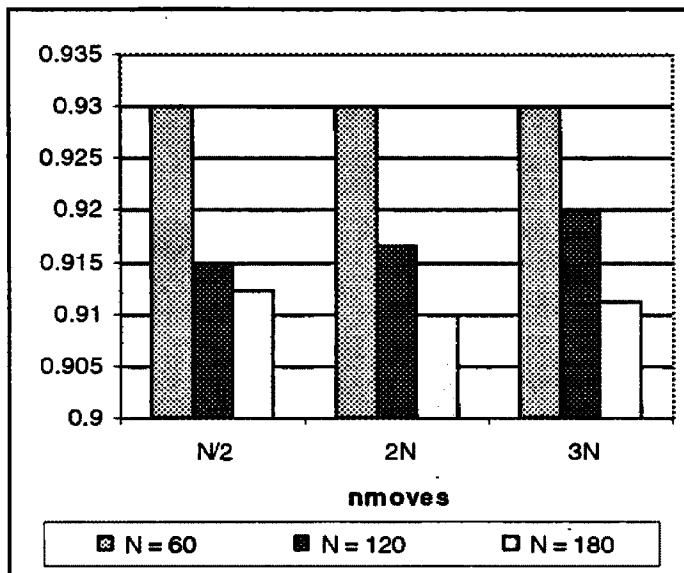


**Figure 5.42.** The trend of an increase of average fitness with increasing  $K$  settings continues on the  $NK$  landscapes with  $MMHC2$ . The  $e$ moves parameter values we used have little effect on this trend.

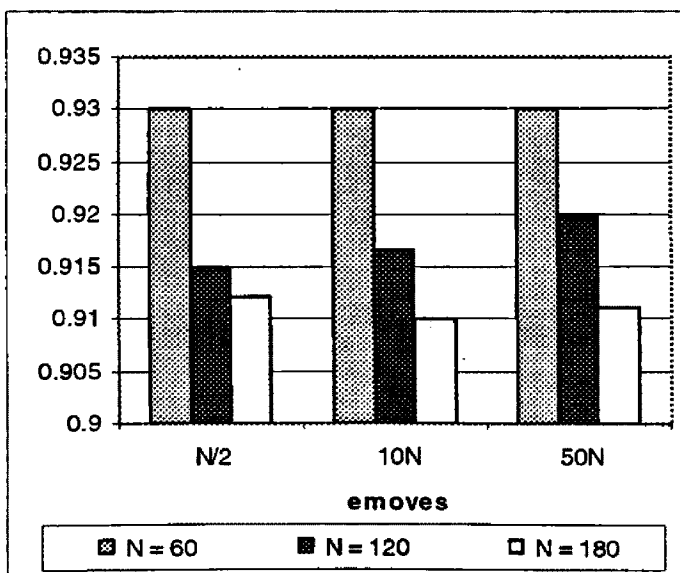
## 5.4 RMHC / All Problem Sizes

The data presented for the *RMHC* algorithm are relatively uncomplicated, due to the fact that for *RMHC*, we change only one bit at a time. Thus, *bits* is set to 1 throughout. On the deceptive and easy landscapes,  $K = 6$  on all problems.

### 5.4.1 RMHC / Deceptive: $bits = 1 / nmoves$ and $emoves$



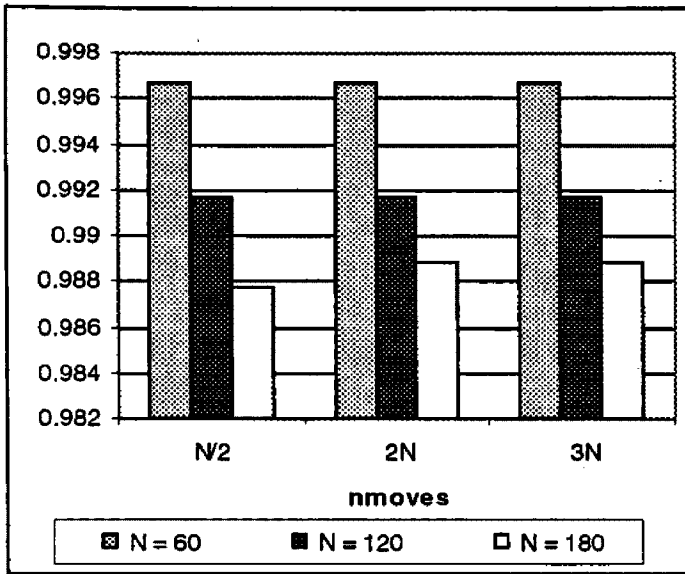
**Figure 5.43.** For all settings of *nmoves*, the *RMHC* algorithm achieves its best results on the 60 bit problems. Changes in *nmoves* result in only small differences in average fitness here.



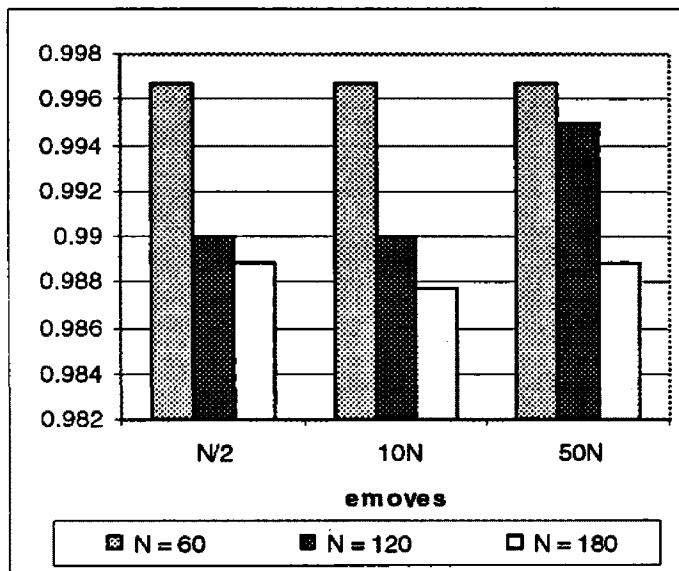
**Figure 5.44.** A similar thing can be said for all settings of the *emoves* parameter with *RMHC* as for the *nmoves* parameter: average fitness is greater on the 60 bit problems. Changes in the *emoves* parameter have a negligible effect on average fitness.



### 5.4.2 RMHC / Easy: bits = 1 / nmoves and emoves



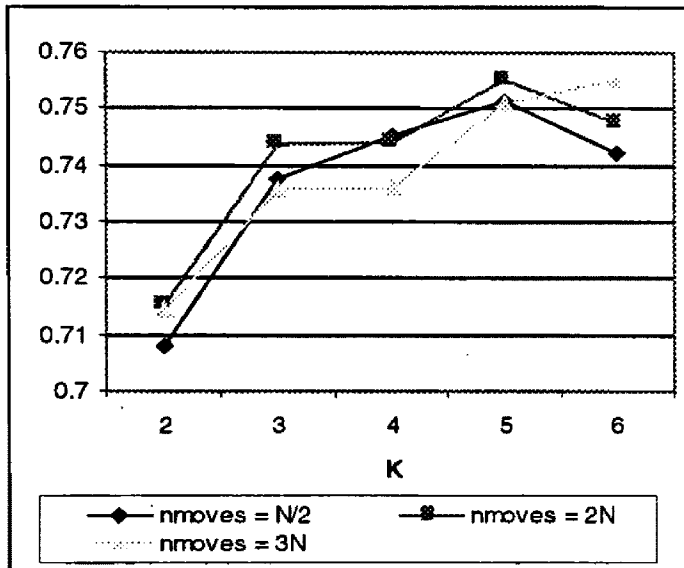
**Figure 5.45.** Average fitness achieved by RMHC on the easy landscape is also highest with the 60 bit problems, although no problem size can be said to do poorly here. Changes to the *nmoves* parameter have negligible effect.



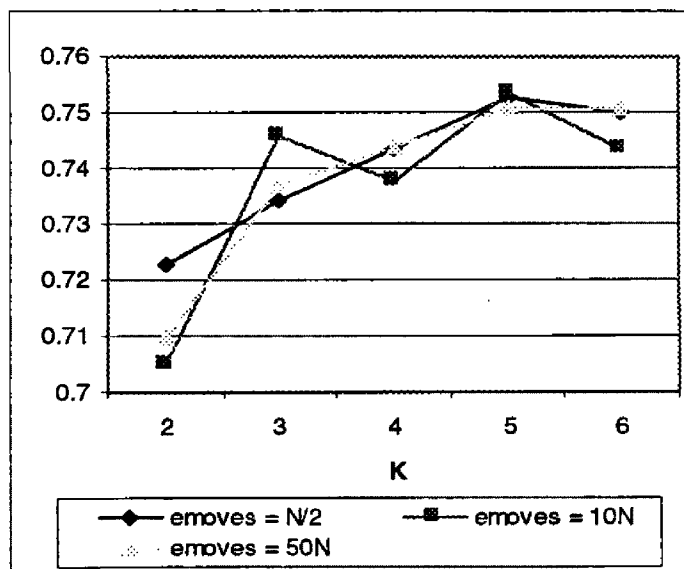
**Figure 5.46.** Very good results are obtained by this algorithm on the 60 bit problems. The *emoves* parameter seems to cause a slight improvement in average fitness on the 120 bit problems.

### 5.4.3 RMHC / NK Landscapes: $bits = 1 / nmoves$ and $emoves$

On the  $NK$  landscapes, we vary the  $K$  parameter, so there is some additional complexity reflected in the data. The  $bits$  parameter is set to 1. Since we do not know the global maximum of the  $NK$  landscapes, these data are more comparative than definitive.



**Figure 5.47.** The RMHC data continue to demonstrate the trend of general improvement in average fitness accompanying an increase in the  $K$  parameter. Changes in  $nmoves$  produce erratic results here, although a beneficial setting for  $nmoves$  can be found for each value of  $K$ .



**Figure 5.48.** The chart depicting the effects of different  $emoves$  settings is very similar to that for  $nmoves$  here. Overall fitness improves with an increase in  $K$ , except for the setting of  $K = 4$  where the fitness decreases slightly for a setting of  $emoves = 10N$ .

# Chapter 6

## Conclusions

Overall, the data obtained from this investigation into the search characteristics of three hillclimbing algorithms tend to support, to some degree, all three hypotheses we made at the outset of the study. We refer to the sections of Chapter 5 containing the charts, for our discussions in the present chapter. The charts are organized into three major categories according to algorithm. Preceding the charts that detail results for the separate algorithms (Sections 5.2, 5.3 and 5.4), there is a section of general overview data encompassing the entire study (Section 5.1).

### 6.1 Superiority of Macromutation

We hypothesized that the macromutation algorithms, *MMHC1* and *MMHC2* would achieve better results than the single-bit-mutation algorithm, *RMHC* on all test problems (landscapes), and on all problem sizes. We further conjectured that macromutation would do especially well on the non-separable functions, the *NK* landscapes. Figures 5.1, 5.2

and 5.3 show the average fitness values achieved for all the algorithms on the fully deceptive, fully easy and NK landscapes, respectively, for all problem sizes combined. The charts in Figures 5.1 and 5.2 demonstrate superior results obtained with the *MMHC2* algorithm for *bits* = 3, 4, or 5 on the deceptive landscape and *bits* = 2 or 3 on the easy problems. The chart in Figure 5.3 shows that *MMHC2* also did well in comparison with *RMHC* and *MMHC1* on the *NK* landscapes. In Figure 5.3, we see that *MMHC1* does better in comparison with *MMHC2* on the non-separable *NK* landscapes than it does on either of the separable, easy or deceptive, landscapes. In Figures 5.1 and 5.2, it can be seen that *RMHC* performs quite well in comparison with *MMHC1* on all the separable functions. *RMHC* actually performs better than *MMHC1* on all runs with the easy problems. While localized macromutation (*MMHC2*) did very well overall, distributed macromutation (*MMHC1*) performed somewhat less well than expected.

## 6.2 Superiority of Localized Macromutation

On the test problems in this work, we have defined the parameter  $K$  to be the length of the substrings, the average of whose individual fitnesses comprise total string fitness. At the outset of this study, we hypothesized that the macromutation algorithm that tries to mutate bits that are coupled in the fitness function, *MMHC2*, would outperform the macromutation algorithm whose bits are more likely to be affine in the fitness function, *MMHC1*. We conjectured that this would be true on all test problems whenever the mutated bits are within the length of one substring of each other, that is, whenever the mutated bits are more likely to be coupled within the fitness function. Multiple mutations along a string are more likely to occur within the length of one substring of each other as

we increase the size of the substring, thus making it inclusive of more bits. An increase in the  $K$  parameter should result in more of the mutated bits being ones that are coupled in the fitness function. We conjectured that as we increase  $K$  we would see a relative improvement in performance for localized macromutation (*MMHC2*). In addition, since the bits mutated by *MMHC2* are more likely to be coupled bits in the fitness function, we expect *MMHC2* not only to do better than *MMHC1* but to increase its relative superiority over *MMHC1* as  $K$  increases. These expectations are shown to be justified by the results on the  $NK$  landscapes, Figure 5.10. (Since we vary the  $K$  parameter only on the  $NK$  landscapes, charts for the  $NK$  landscapes are the only ones that include changes in the  $K$  parameter.) Since we do not have data concerning the global maximum of each of the  $NK$  landscapes, we are not able to definitively specify the algorithms' *absolute* performances with the  $NK$  functions.

### 6.3 The Strength of *RMHC*

The single-bit-flipping hillclimber, *RMHC*, also fulfills most expectations in this investigation. We originally hypothesized that *RMHC* would perform better on the separable easy and deceptive functions than it performs on the non-separable  $NK$  functions. Since we have no data on the global maximum for the  $NK$  landscapes, we cannot definitively describe *RMHC*'s *absolute* performance there. However, we can see in Figure 5.3 that in comparison with the other algorithms, *RMHC* performs somewhat less well on the  $NK$  problems than either *MMHC2* or *MMHC1* with most *bits* settings. On the deceptive problems (Figure 5.1), *MMHC2*'s overall superiority notwithstanding, *RMHC*, at *bits* = 1, performs better than *MMHC1* does for most of its own *bits* settings.

*RMHC*'s greatest strength seems to lie in its ability to effectively find local maxima. Its outstanding performance on the easy problems (Figure 5.2), especially in relation to *MMHCI*, is indicative of this ability. *RMHC* significantly outperforms *MMHCI* for all of *MMHCI*'s *bits* settings on the easy landscape. Due to *RMHC*'s mixed results on the separable landscapes, we cannot conclude that separability is the deciding factor on this hypothesis.

#### **6.4 The effects of varying *n*moves, *e*moves and *N***

At the outset of this investigation, we sought to observe the influences on algorithm performance of the *n*moves and *e*moves parameters. We also wished to observe the effect of changes in problem size, *N*. We believed that varying these parameters would likely result in changes in algorithm performance. Each of the major sections in chapter 5 contains subsections that detail the effects of these parameter variations with the different algorithms on the three landscapes that we studied. We summarize those data in the following sections.

##### **6.4.1 *n*moves and *e*moves**

We conclude that variations in the *n*moves parameter, the maximum times a new point can have lower fitness than a current point, have a greater effect on overall algorithm performance than do variations in *e*moves, the maximum number of equal fitness steps. We clearly see this by comparing the *n*moves and *e*moves overall data charts for each landscape in Subsections 2 and 4 of Section 5.1. Similar results throughout the study can be seen in comparable charts for every algorithm. From these results, we conclude that a

setting of *nmoves* of  $2N$  or  $3N$  in this study contributes more to good algorithm performance than does the  $N/2$  *nmoves* setting. We also conclude that the two higher *nmoves* settings result in better performance than any of the three settings of *emoves* that we tried. The contribution of *nmoves* to the performance of all algorithms seems especially pronounced on the easy and deceptive separable functions, with the macromutation algorithms, as seen in Subsections 5.2.2, 5.2.5, 5.3.2 and 5.3.5. In almost every chart in Chapter 5 that involves the *emoves* parameter, it seems clear that performance differences are negligible for changes in *emoves*.

#### 6.4.2 $N$

Comparative average fitness differences seemingly related to differences in problem size are apparent throughout this investigation. The algorithms we study seem to achieve higher average fitness values with problem size  $N = 60$  on all landscapes. This is suggested by the data in Figures 13, 16, 17, 18, 21, 22, 28, 31, 32, 33, 36, 37, 43, 44, 45 and 46 of Chapter 5. Once again though, since we do not have data concerning the global maximum of the  $NK$  landscapes, we are not able to definitively specify the algorithms' *absolute* performances there. With that caveat in mind, we cannot conclusively attribute improved performance at  $N = 60$  to problem size. We suggest that it may be due to the simple fact that shorter problems are probably easier to solve.

### 6.4.3 The *bits* Parameter

Throughout the data produced by this investigation, it can be seen in most of the charts involving the *bits* parameter with the MMHC2 algorithm, a setting of  $bits = 3$  seems to produce exceptional results, comparatively. We suggest that this improvement is due to MMHC2's potentially mutating  $2 * 3 = 6$  bits simultaneously.

## 6.5 Summary

Throughout this study, we have seen that hillclimbing algorithms using macromutation perform very well on a variety of landscapes. We have also observed that this is emphatically demonstrated for algorithms whose mutated bits are coupled within the fitness function. We have obtained improved results on the particular problems studied whenever we allow the algorithm to have  $2N$  or  $3N$  retries in seeking improved steps whenever it finds less fit points during a search. In most cases, the setting of  $nmoves = N/2$  seems to degrade performance. Additionally, we have seen that allowing our algorithms to explore mesas on the landscapes tested results in no significant increase in performance. The algorithms in this study return somewhat higher average fitness values with a problem size of 60 bits, most likely because the longer problems are harder to solve.

At this time, we present an answer to the question, "Why does macromutation do well on separable functions?" It is clear from these results that macromutation does well on separable functions. We state that this is because macromutation, by its nature of simultaneously mutating multiple bits, has a high likelihood of mutating bits that are



coupled in the fitness function. Bits that are coupled in the fitness function are more likely to produce a greater increase in fitness than mutating bits that are affine in the fitness function. In the separable functions used here, the fixed value of  $K = 6$  increases the likelihood that the bits mutated will be coupled in the fitness function.

# Appendix

**Figure A.1.** Sample UNIX C-Shell script file named M2, used to run the *MMHC2* algorithm on the final set of hillclimbs.

```

# driver script for mmhc2
if ( $#argv != 2 ) then
  echo "USAGE: M2 <evals> <seed>"
  exit
endif
set evals = $argv[1]
set seed = $argv[2]
foreach N ( 60 120 180 )
  @ nh = $N / 2
  @ n2 = 2 * $N
  @ n3 = 3 * $N
  @ n10 = 10 * $N
  @ n50 = 50 * $N
  foreach K ( 2 3 4 5 6 )
    foreach bits ( 2 3 4 5 6 )
      foreach nmv ( $nh $n2 $n3 )
        foreach emv ( $nh $n10 $n50 )
          @ seed++
          mmhc2 $N $K $bits $evals $nmv $emv nk $seed
          if ( $K == 6 ) then
            @ seed++
            mmhc2 $N $K $bits $evals $nmv $emv e60 $seed
            @ seed++
            mmhc2 $N $K $bits $evals $nmv $emv d60 $seed
          endif
        end
      end
    end
  end
end
end
end
end
end

```

**Figure A.2.** Sample UNIX command-line invocation of the *MMHC1* algorithm for performing the initial, exploratory hillclimbs. A line of comments above the command line itself identifies the parameters.

```
#      N   K  #bits #climbs  nomovemax  emovemax  landscape  seed#
mmhc1 60  6   4     10     120         300       e60        102938
```

**Figure A.3.** Sample UNIX command-line invocation of the *MMHC2* algorithm for performing the final set of hillclimbs. A line of comments above the command line itself identifies the parameters.

```
#      N   K  #bits  #evals  nomovemax  emovemax  landscape  seed#
mmhc2 120  6   4     5700000  120         300       d60        107401
```

# Bibliography

- [1] Kihong Park. A Comparative Study of Genetic Search. *Proc. 6th International Conference on Genetic Algorithms*, July, 1995, 512-519.
  
- [2] M. Srinivas and Lalit M. Patnaik. Genetic Algorithms: A Survey. *Computer*, June, 1994, 17-26.
  
- [3] Terry Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquerque, New Mexico, May, 1995.
  
- [4] Stuart Russell and Peter Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
  
- [5] Stuart A. Kauffman. *The Origins of Order; Self-Organization and Selection in Evolution*. Oxford University Press, New York, 1993.
  
- [6] Richard K. Thompson and Alden H. Wright. Additively Decomposable Fitness Functions. Paper. The University of Montana, Missoula, January, 1997.
  
- [7] K. Deb and D. E. Goldberg. Sufficient Conditions for Deceptive and Easy Binary Functions. Technical report, University of Illinois, Champaign-Urbana, 1992. IlliGAL Report 92001. Available by ftp from gal4.ge.uiuc.edu in pub/papers/IlliGALs/92001.ps.Z