2011

Establishing parameters for problem difficulty in permutation-based genetic algorithms

Adam Nogaj

Establishing Parameters for Problem Difficulty in Permutation-Based Genetic Algorithms

by

Adam F. Nogaj

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science
in Computer Science

Supervised by

Professor Dr. Roger S. Gaborski
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York
October 2011

Approved by:

Dr. Roger S. Gaborski, Professor
Supervisor, Department of Computer Science

Dr. Peter G. Anderson, Professor Emeritus
Reader, Department of Computer Science

Dr. Stanislaw P. Radziszowski, Professor
Observer, Department of Computer Science

Dedication

To my parents, Larry and Colleen Nogaj,
to my grandparents, Wilfred and Susan Finn, and Stan and Rena Nogaj,
and to my amazing wife, Lisa,
who have surrounded me with more kindness, support, generosity, and love
than I could ever begin to describe.
I love you all.

Acknowledgments

First and foremost, I would like to thank RIT Department of Computer Science Graduate Coordinator Dr. Hans-Peter Bischof and my thesis committee, especially Dr. Peter Anderson, for their ongoing patience and willingness to help guide and support me throughout my thesis. I would also like to thank the math and computer science faculty at SUNY Fredonia, especially Dr. Ziya Arnavut, for encouragement of my intellectual pursuits as an undergraduate student.

Abstract

Establishing Parameters for Problem Difficulty in Permutation-Based Genetic Algorithms

Adam F. Nogaj

Supervising Professor: Dr. Roger S. Gaborski

This thesis examines the performance of genetic algorithm (GA) crossover techniques within two problems: n-queens with poison (NQWP) and processor scheduling (PS). Each problem was analyzed at sizes of 32, 64, and 128, referring to number of queens to be placed and number of single-time-unit processes to be scheduled, respectively. The specific crossover techniques studied were cycle crossover, order crossover, partially mapped crossover, merging crossover, and one-point, two-point, and uniform signature representation crossover, in addition to various greedy approaches. In conjunction with tests that vary crossover techniques, experimentation was performed to determine what percentage of problem constraints (poisoned squares for NQWP or precedence relationships between tasks for PS) makes the problems most difficult to solve, that is, the constraint densities at which optimal solutions require the highest number of GA fitness evaluations. While minor fluctuations in difficulty occur upon variations in fitness function and problem size, the NQWP problem is most difficult around a constraint density of 0.8 and the PS problem is most difficult around constraint densities of 0.2 to 0.3. Even within an individual problem, one crossover technique does not irreproachably outperform others. However, cycle crossover stands out in its performance in the PS problem while merging crossover and uniform signature crossovers most often perform well for NQWP.

Contents

Chapter 1

Introduction

1.1 Genetic Algorithms

A genetic algorithm (GA) is a problem-solving methodology based on biological evolutionary principles. Through high school biology classes, evening news segments depicting modern advances in health and technology, and casual references in popular culture, there exists a virtually ubiquitous basic understanding of the role of that DNA plays in determining a person's (or other organism's) seemingly countless characteristics. Combinations of just four chemical compounds within strands of DNA can contain biological instructions that influence height, eye color, and even personality traits [6]. A different combination of the same four chemicals at a different part of the DNA can predispose an individual to a particular disease [7], among many other things. Additionally, it is well-understood that an offspring's DNA is created exclusively by taking portions of each parent's DNA in a largely randomized process termed chromosomal crossover [8]. An individual's DNA is also susceptible to a phenomenon called mutation during which it becomes altered (damaged) just slightly [9].

Further, most are at least as equally familiar with the basics of the Darwinian principle of *survival of the fittest* which states that individual organisms with desireable or

beneficial characteristics will have the greatest opportunity to survive, and therefore, ultimately the greatest opportunity to reproduce. Through such procreation, an individual's characteristics (DNA) are passed on to children, and more generally, it suffices to say that those characteristics have an increased likelihood of being exhibited in the overall population. The prevalence of such characteristics may then increase further via continued biological reproduction [10].

In GAs, an individual's DNA represents information sufficient to describe a potential solution to a problem. It is often generalized that a bit string [2] (series of 1s and 0s) may be used as the language of GA DNA since at some level, anything can be represented by such binary primitives. However, realistically, any standardizable language may be utilized within a GA individual's DNA. For example, consider the problem of getting from one's house to a particular grocery store, such as the Wegmans in Pittsford, NY. DNA in this problem may be a set of instructions to be carried out at a given intersection, such as *turn left*, *turn right*, and *go straight*. It is easy to see that any list consisting of left turns, right turns, and go-straights does constitute a reasonable set of instructions to attempt to direct an individual toward the grocery store, even if it were randomly generated or otherwise not a very good solution. That is, there are no out of place or meaningless instructions such as *bake cookies* or *throw a Frisbee*. And if it is to be assumed that any intersection is a standard three or four-way intersection, then this language of such driving maneuvers is entirely sufficient to describe any necessary directions. In general, data that can be easily represented in an array or list is an ideal format for GAs [2], analogous to how DNA is essentially an ordered list of chemical compounds.

It is of course unlikely that a randomly created set of such instructions will actually lead one to the Pittsford Wegmans. That said, it is not difficult to come up with a metric to describe the quality of the destination reached for a given set of instructions,

randomly generated or otherwise. One immediately reasonable idea would be to calculate the straight-line distance from the destination reached after the instructions complete to the actual desired destination, and the lower the distance, the better the solution. However, this method does not take gasoline or time required into account; a set of directions that may lead one a thousand miles off course before reaching the destination would be considered to have equal quality as an intelligent set of directions that minimizes distance traveled. Therefore, it may make more sense to create a somewhat more complex evaluation of a solution that considers both miles traveled and closeness to the desired destination. Through creating a series of numeric penalties and/or rewards for certain solution characteristics, this problem could be further expanded into a decision algorithm intended to take one to a nearby grocery store, with a particular preference toward Wegmans, and especially the one in Pittsford, NY if it can be reached within a reasonably short drive. Assigning penalties and/or rewards such that this becomes a successful algorithm may take several attempts at parametrization, not to mention what constitutes a successful algorithm may be a very subjective thing to begin with.

Regardless, such a quality-analysis of a potential solution is referred to as the GA fitness function. Devising the fitness function and deciding upon an individual's DNA representation comprise the two most problem-specific (and likely most important) decisions one must make when implementing GAs [2]. Note that one need not know the actual ideal solution to the problem to have the ability to gauge the quality of a proposed solution via a fitness function. This is a fundamental tenet of GAs. A GA relies on the ability to easily create a well-formed potential solution and also determine the quality of such a solution; it is not necessary to have any particular information about what specifically might make a solution high-quality. (For example, in the above situation, all of the best solutions may end up avoiding a particular intersection

even though such avoidance was not an actual constraint of the problem.)

With the concepts of DNA representation and fitness function in place, it is now possible to describe the GA process in full. For any step of the process, there may additional variations and options which may alter (and hopefully improve) GA performance. Additionally, the designer of a particular GA also certainly has the option to implement novel features which may be specific to the problem at hand. As such, to convey general behavior, only some of the most basic and general principles shall be discussed in the following example GA methodology [2] [1]:

[1] Specific parameters and options actually utilized within this thesis are described in the following chapters and sections.

Algorithm 1.1 Example GA Methodology

1) Create a population of n individuals, each with initially random DNA.

2) Compute and store the fitness of each individual.

3) Select two individuals (parents) for reproduction, generally weighted so that individuals with higher fitness have a greater chance of selection.

4) Create m individuals (children) from the parents by taking some genetic information from one parent and the rest from the other. (In a standard array-based representation, one-point crossover is common: given DNA of length k, randomly choose c, $0 \leq c < k$; the DNA in a child will contain all values in the range $(0, c]$ from the first parent and all values in the range (c, k) from the second parent.)

5) Allow for the possibility of mutation, that is, for the possibility of atomically small portions of the DNA to alter values arbitrarily. (This may be accomplished by performing such a partial re-randomization if a particular randomized number is sufficiently small.)

6) Compute and store the fitness of each child created though such crossover.

7) Create space in the population for the children by selecting m individuals to remove from the general population, generally weighted so that individuals with lower fitness have a greater chance of deselection. (A simple and often effective method is to merely deselect the worst m individuals.) Insert the children in place of the deselected individuals.

8) Repeat steps 3 through 7 until a particular benchmark is reached: If the fitness function is designed such that there is a perfect fitness value, the GA should terminate upon finding such a perfect individual. However, if it is not possible to determine perfect fitness, or to simply prevent the GA from running indefinitely, it should also terminate if a predetermined limit of reproductions or runs of the fitness function has been reached.

9) In cases where the performance of the GA is of interest (in addition to or instead of the actual solution, as is the case with the research contained within this thesis), the GA should also report the number of fitness evaluations performed upon termination. This then generally signifies the number of fitness evaluations required to find the perfect solution or indicates that the GA did not solve the problem within the predetermined fitness evaluation threshold.

The crux of the GA performance rests with the hope that after continually selecting relatively high-quality individuals for reproduction, occasionally even higher quality children are inserted back into the population. This behavior, over time, ideally leads to the creation of a perfect solution [2].

GAs have been shown to achieve quality results in the following areas [11], amongst others:

- Antenna design

- Drug design
- Chemical classification
- Electronic circuits
- Factory floor scheduling
- Turbine engine design
- Crashworthy car design
- Protein folding
- Network design
- Control systems design
- Production parameter choice
- Satellite design
- Stock/commodity analysis/trading
- VLSI partitioning/placement/routing
- Cell phone factory tuning
- Data Mining

1.2 Permutation-Based Genetic Algorithms

A permutation-based genetic algorithm is a genetic algorithm where an individual is represented by a permutation, as opposed to a bit string or array of arbitrary numbers, etc. There are many problems (in addition to the ones selected for this thesis)

highly amenable to a permutation representation, perhaps most notably the traveling salesman problem (TSP). In general, situations that can be easily expressed as a specific ordering of entities are particularly well-suited for this type of representation.

Special attention must be given to crossover methods applied to permutation-based individuals. In the traditional sense, a crossover would simply create a new individual by preserving exact location of some genetic information from each parent. However, that naive process cannot be applied to permutations, which by definition require that each integer value in $[0,n)$ appears exactly once in the permutation. Therefore, permutation-specific crossover approaches must be considered to correctly and legally commingle genetic data of two parent individuals so that crossover of two permutations also results in a permutation. While any such algorithm that results in a true permutation constitutes a legal permutation crossover method, specific ones have been chosen for application in this thesis; they are discussed in 2.1.

1.3 Ordered Greed

Ordered greed (OG) signifies a general methodology which can be applied in different circumstances. According to Anderson and Ashlock (2004),

> Ordered Greed is a form of genetic algorithm that uses a population of permutations whose fitnesses depend on their use as the orders in which parts of a problem are solved. For example, a permutation may specify the order of the rows in which chess-board queens are placed to try to avoid attacks by other queens, or it may specify the order in which vertices of a graph are colored to avoid adjacent vertices getting the same color. The problems mentioned are surrogates for practical, difficult, real-life problems such as scheduling.[1]

Precise ordered greed implementations for the problems studied within this thesis will be described in Chapters 3 and 4. However for an immediate example, consider the potential application in graph coloring as noted above. In a small problem, [2 4 1 3 0] would suggest the following coloring strategy. Assuming vertices v_0 through v_4 and an arbitrarily-sized set of colors $C = \{c_0, c_1, \ldots\}$, color v_2 with the first (lowest-numbered) available color (c_0). Next, color v_4 with the first available color that will not conflict with anything previously colored (so far, just v_2). Specifically, if (v_2,v_4) is in the graph's edge set, then the first available color is c_1. Otherwise, it can be colored with c_0 as well. Repeat this process for the remainder of the permutation.

1.4 Additional Definitions and Clarifications

1.4.1 Crossover Techniques

The terminology *crossover techniques*, *crossover methods*, and *crossover algorithms* may be used interchangeably within this thesis.

Some crossover techniques (as described in Section 2.1) may be defined differently in other texts. Such differences could potentially lead to very non-trivial in their behavior, compared to varieties used in this thesis.

1.4.2 Problems, Runs, Tests, and Experiments

A *problem* refers to a potential or actual application of GAs. For example, in this research, the n-queens with poison and processor scheduling problems are studied at length.

A *run* of a GA is typically a single GA execution, from start to finish, that outputs the number of fitness evaluations required to solve the problem for a given set of parameters and characteristics.

Tests and *experiments* are used interchangeably to refer to sets of related runs.

1.4.3 Difficulty

Within this thesis, *difficulty*, and in general, its traditional synonyms and antonyms, pertain to the number of fitness evaluations required to solve a problem. That is, a difficult (or hard, etc.) instance of a problem requires a comparatively large number of fitness evaluations to solve, and an easy (or simple, etc.) problem requires relatively few fitness evaluations. Further, some problems may be said to be easier or harder to solve than others, and this too is just another relative comparison of fitness evaluations needed for solutions.

1.4.4 Constraint Density

To aid in the discussion of where problems are difficult, the terminology of *constraint density* will be used. Constraint density refers to the percentage of actual constraints upon a particular data set, where 0.0 implies that there are no constraints on the data set and 1.0 implies that every possible constraint on a data set is in place except for the actual solution (that is, the data set cannot be constrained to the point where there is no solution). While constraint density will also be discussed later (and further) for particular specific data sets and problems (in Chapters 3 and 4), for a brief, general, and unrelated example, consider the following situation.

A teacher is in the process of creating a new seating chart for his or her students, all of which sit in a rectangular grid of desks. The teacher decides to accept every mutual request from students wishing to stay seated next to each other under the new seating arrangement. An unconstrained data set (where constraint density equals 0.0) would be a situation where there are no student requests; the teacher can seat students anywhere. A fully constrained data set (where constraint density equals 1.0) would

be one where every student wishes to keep the same neighbors on each side. One may notice, under the definitions of this problem, that the teacher need not keep the exact same seating chart in this scenario; rows would be interchangeable and within each row, the order could be reversed. A constraint density of 0.5 would imply that 50% of the possible pairings are requested. Of further note, the set of possible constraints is clearly just a subset of possible relations between the students; however, by the definition of the problem, relations that are not in the possible constraint set are not considered at any point in the problem. For example, the teacher does not take student requests to maintain neighbors in front or behind, nor does the teacher take requests to sit next to an arbitrary student.

1.5 Practical Approach

Even while it may only take a matter of several seconds for GA to solve a single instance of a problem in the worst cases, the number of GA runs required to perform research quickly becomes enormous. Considering the parameters and options that pertain to problem type, problem size, constraint density, crossover technique, population size, population representation, selection method, deselection method, mutation rate, and potentially other characteristics, it is sufficient to say that there are many dimensions across which GA performance can be measured. Further, given the random nature of GAs, it is scientifically necessary to run a single experiment many times to better determine the general behavior associated with a particular combination of parameter values. Additionally, in this research, a particular constraint density only specifies an amount of constraints (and not where or how a problem is specifically constrained). Therefore, a specific constraint density must be applied several times to learn the general behavior associated with only that number of constraints.

To focus on GA behavior and performance within the scope of this thesis, variations

of population representation, selection and deselection methods, and mutation rates were not actively studied. Additionally, other paramaters were at times not varied greatly, even where there may have been interest in such greater variation. Such paring down of parameters allowed for stages of research to explore new ideas based on current results that would not have been possible if one blanket set of parameter ranges was utilized throughout. Therefore, there may be immediate room for future research simply via measuring GA performance under ranges of seemingly standard parameters not explored within this thesis.

Descriptions of parameters and characteristics that were actively studied follow in the next chapter.

Given the computation-intensive nature of this research, experiments were run on several computers. This was coordinated via a central FTP-based solution where participating computers downloaded tasks (as a specific combination of parameters), ran the GA under such parameters, and reported the results back to the main FTP server. In total, up to six computers with fourteen processors between them worked on the experiments within this thesis. However, as these computers were, in general, multi-use personal computers, they experienced various stretches of downtime or shared processor use.

Even with such extended computing resources, many of the experiments within this research still took several days (and up to roughly a week) to complete.

Of further note, while mechanisms were in place to maintain the integrity of this automated distributed system, at a very low rate, individual tests at times did not properly download or complete, or their associated results did not properly upload. On occasion, the same task was taken on by more than one computer as well. The net result of such behavior is that while the intention was to run a GA on any given set of parameters k times (for some value of k), there existed instances where due

to such errors, a particular set of parameters was run slightly more or slightly fewer than k times. Such possible discrepancies were accounted for in analysis of results, and further, the few scattered missing tests did not hinder or complicate any such analysis.

1.6 Visual Representation of Data

In this thesis, data is most often represented in a chart with colored backgrounds to aid in the visual representation of information. This depiction style allows for a larger amount of data to share a physical space in a clearer manner than, for example, a graph with many lines and data points which may become cumbersome to read, especially when data lies near each other. Figure 1.6.1 describes explanation of notation, data representation, and where to find pertinent information.

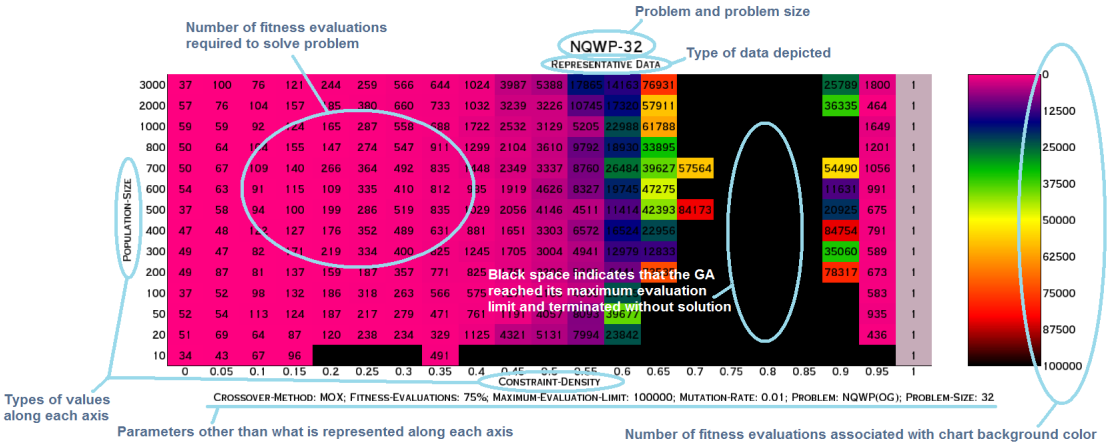

Figure 1.6.1: Explanation of chart format

Chapter 2

Techniques

2.1 Crossover Methods

The following crossover techniques are utilized in this research and described in this chapter:

- cycle crossover (CX)
- order crossover (OX)
- partially-mapped crossover (PMX)
- merging crossover (MOX)
- signature representations (SX)
  - 1-point
  - 2-point
  - uniform
- hybrid of CX/OX/PMX/MOX (HX)

CX, OX, and PMX have been chosen largely due to their ubiquity as crossover methods. MOX has been selected because of how it is generally amenable to ordered greed

approaches and SX has been selected because of its similarities to standard crossover techniques of non-permutation-based strings. A hybrid technique was examined to gauge any potential benefits of maintaining a variety of different crossover methods.

2.1.1 Cycle Crossover (CX)

Given permutations (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm[4]:

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) Randomly select $i_{current}$ such that $0 \leq i_{current} < n$, assign $i_0 \leftarrow i_{current}$

3) Mark index $i_{current}$ for crossover

4) Select i_{next} such that $A[i_{next}] = B[i_{current}]$

5) Assign $i_{current} \leftarrow i_{next}$

6) Repeat steps 3 through 5 until $i_{current} = i_0$

7) Exchange values of A' and B' on marked indexes

The purpose of the cycle found in steps 2 through 6 is to ensure that an equal set of values is to be exchanged between both permutations and that these values are to be exchanged such that no other parts of the permutation are altered in the crossover process. It can therefore be said that CX is a crossover technique that aims to be successful by preserving positions of values.

Of particular note is that the length of the cycle cannot be predicted, and therefore the same can be said about the amount of differentiation between parents and children. Further, in two worst case scenarios, it is possible that children are exact copies of parents: when $A[i_0] = B[i_0]$ the cycle is of length of 1 and no differing values are

swapped, as well as when the cycle spans the entirety of the permutations (a cycle length of n) as that results in all values being swapped, in essence merely relabeling each child in lieu of exchanging any values.

Figure 2.1.1 depicts an example of the CX crossover process.

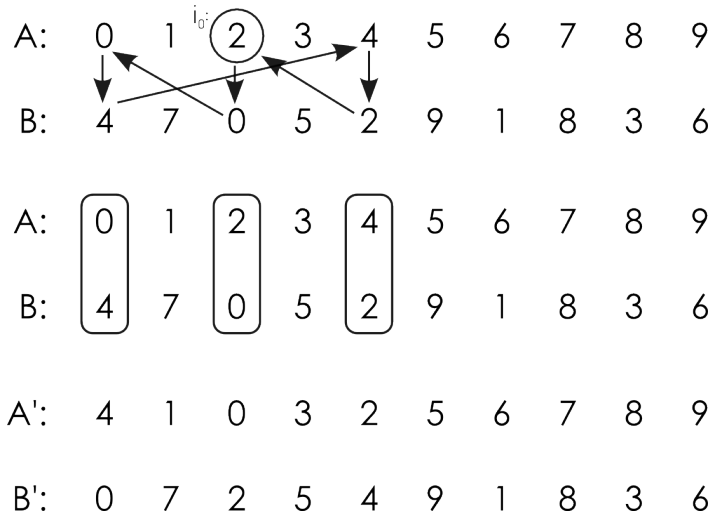

Figure 2.1.1: Example of CX algorithm

2.1.2 Order Crossover (OX)

Given permutations (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm [2]:

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) Randomly select p_1 and p_2 such that $0 \leq p_1 \leq p_2 < n$

3) For each index i such that $p_1 \leq i \leq p_2$, swap $A'[i]$ and $B'[i]$

4) For each child A' and B', if a value within the swapped portion from step 3 appears elsewhere in the permutation, remove it elsewhere in the permutation and temporarily leave that space blank

5) For each child A' and B', starting immediately after p_2 (and continuing at the beginning of the permutation if necessary) left-shift all remaining values and blanks in the permutation so that any blank spaces appear immediately before p_1 (and at the end of the permutation if necessary)

6) For each child A' and B', starting immediately after p_2 (and continuing at the beginning of the permutation if necessary) fill in blanks with values not currently represented in the permutation, in order of how they originally appeared in A and B respectively

OX preserves a contiguous section of a permutation in the resulting children, as shown in steps 2 and 3 above. OX then retains the order of the rest of the permutation, with respect to where those values appeared in the parent permutations. However, since this order begins after p_2 in the child permutations and potentially wraps around to the beginning of the permutation, OX does not necessarily preserve order in the truest sense. When p_1 and p_2 are selected uniformly, the average size of the contiguous crossover section will be roughly $\frac{n}{3}$ for large enough values of n. This derivation is shown in Appendix Section A.

Figure 2.1.2 depicts an example of the OX crossover process.

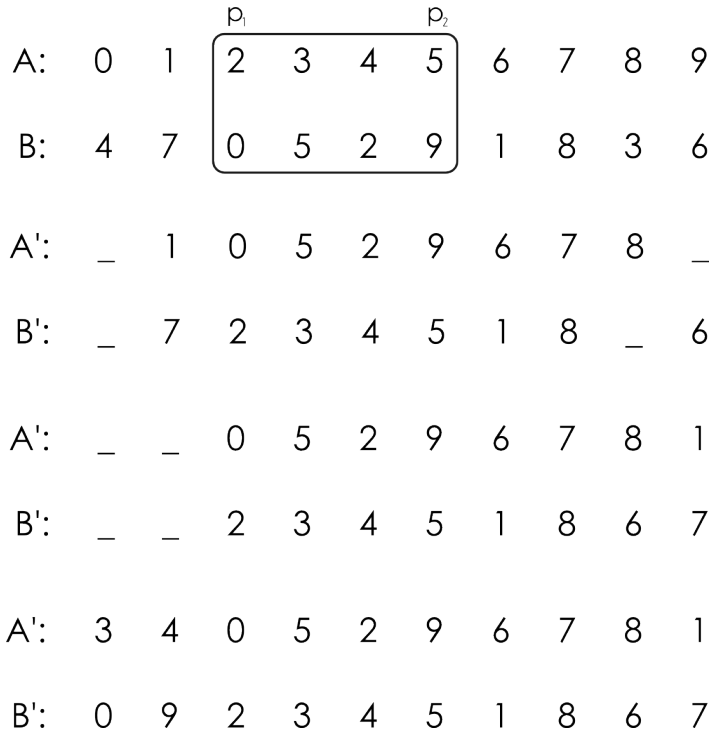

Figure 2.1.2: Example of OX algorithm

An additional variation of OX will be considered. This variation differs from the above algorithm in the following ways:

A) Instead of swapping a contiguous section of the permutation in steps 2 and 3, a certain number of values at individual positions are swapped

B) Instead of filling in values in the order they appeared starting after p_2 as is done in steps 4 through 6, values are filled in starting at the beginning of the permutation ($i = 0$)

2.1.3 Merging Crossover (MOX)

Given permutations (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm [1]:

1) Randomly merge A and B into a list L

2) Let A' be the permutation obtained by taking the first instance of each distinct value in L, preserving their order, and let B' be the permutation obtained by taking the second (last) such instance of each value in L, again preserving order

Notice that MOX does not necessarily preserve the location of any elements. It does however preserve order in the sense that if value x precedes value y in both A and B, x will also precede y in both A' and B'[1]. More generally, it also has the characteristic that when L is the result of relatively uniform merging (constructed without exceptionally long contiguous sections from either parent), values that appear in the same vicinity in both parents will also appear in the same vicinity in both of the children.

Figure 2.1.3 depicts an example of the MOX process.

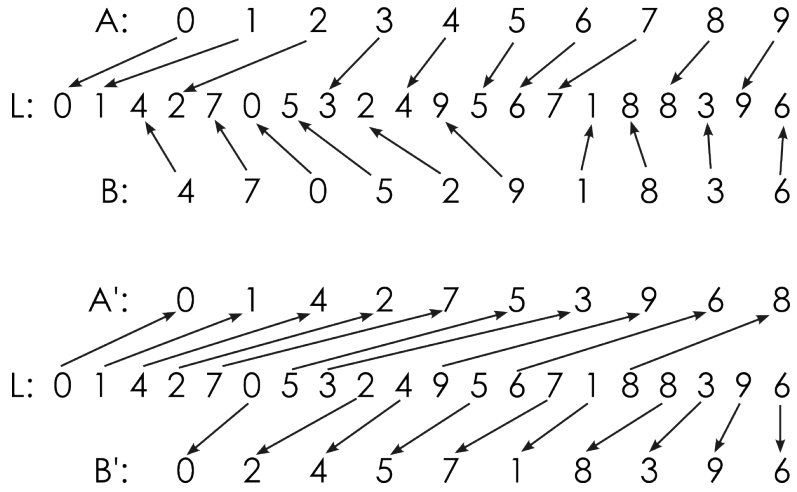

Figure 2.1.3: Example of MOX algorithm

2.1.4 Partially-Mapped Crossover (PMX)

Given permutations (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm (adapted from *Introduction to Genetic Algorithms* by Sivanandam and Deepa [2] with the exception of step 4, which was not included in the cited text [1]):

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) Randomly select p_1 and p_2 such that $0 \leq p1 \leq p2 < n$

3) For each child A' and B', create respective lists of mapping relationships M_A and M_B, where for each index i such that $p_1 \leq i \leq p_2$, $B'[i] \rightarrow A'[i]$ is added to M_A and $A'[i] \rightarrow B'[i]$ is added to M_B

4) For each mapping list M_A and M_B, if there are cycles such that $x \rightarrow z$, and $z \rightarrow y$, replace those two such relationships with a single mapping $x \rightarrow y$; repeat until no such cycles exist1

5) For each index i such that $p_1 \leq i \leq p_2$, swap $A'[i]$ and $B'[i]$

6) For each index i such that $0 \leq i < p_1$ or $p_2 < i <$n, if the value $A'[i]$ begins a mapping in M_A, replace it with the respective mapped value and if the value $B'[i]$ begins a mapping in M_B, replace it with its respective mapped value as well

PMX preserves a contiguous section of a permutation in the resulting children, as shown via steps 2 and 5 above. It also, in general, can potentially preserve the

[1] This necessary step is often omitted in various descriptions of this algorithm (including, for example, *Introduction to Genetic Algorithms* by Sivanandam and Deepa [2] and a selected lecture [4]), assumably due to unintentional simplicity of examples accompanying the algorithm. Following this algorithm without such checks for cycles can (and likely will, at least eventually) result in permutations that repeat values: upon careful inspection, it should be evident that this will occur when there is at least one value x that appears in both A and B in the range $[p_1, p_2]$, but at different positions in the permutation.

location of many of the values outside of the contiguous crossover section. Specifically, any value not mapped by steps 3 and 4 above will remain unchanged in the child permutations. When values must be mapped in the child permutations as per step 6, the resulting set of values do not preserve any particular ordering from either parent. When p_1 and p_2 are selected uniformly, the average size of the contiguous crossover section will be roughly $\frac{n}{3}$ for large enough values of n. This derivation is shown in Appendix Section A.

Figure 2.1.4 summarizes the PMX process.

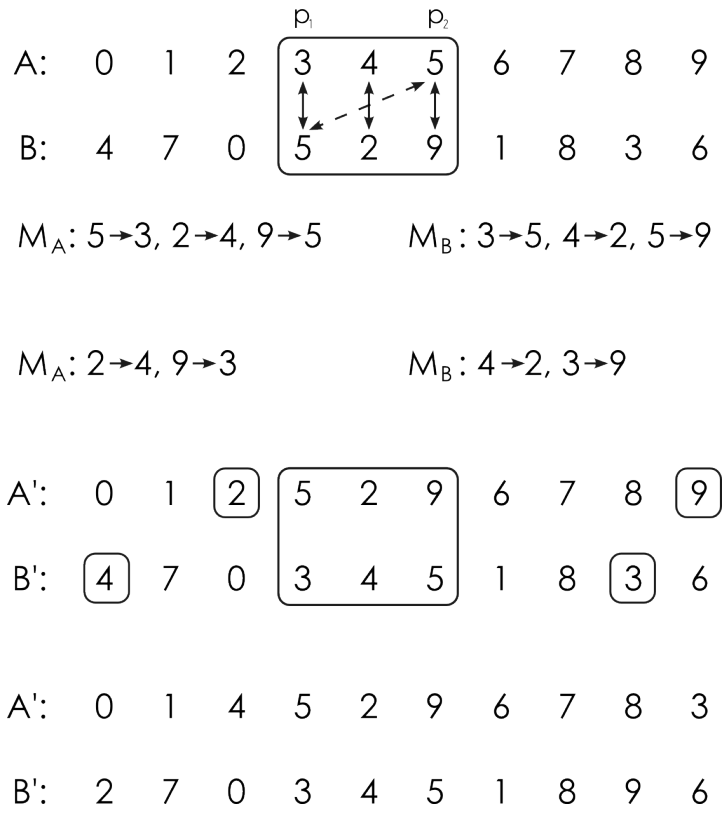

Figure 2.1.4: Example of PMX algorithm

An additional variation of PMX will be considered, as described by Anderson and Ashlock [1].

2.1.5 Signature Crossover (SX)

Signature crossover (SX) is a fundamentally different type of crossover than the previously described methods. In SX, a permutation is instead represented by a signature S of length n created such that for all i, $0 \leq i < n$, $0 \leq S[i] < n - i$. There are $n!$ unique signatures of length n, just as there are $n!$ unique permutations of length n. The use of SX allows for standard (and desirably) straightforward one-point, two-point, and uniform crossover, among other options. A signature S can be converted uniquely into a permutation P for fitness evaluation by the following simple, $O(n)$ algorithm [1]:

1) For all i, $0 \leq i < n$, $P[i] = i$

2) For all i, $0 \leq i < n$, swap $P[i]$ and $P[i + S[i]]$

Figure 2.1.5 below shows a step-by-step application of this conversion algorithm.

S:	4	6	2	2	0	4	1	1	1	0
S[0] = 4, P:	0	1	2	3	4	5	6	7	8	9
S[1] = 6, P:	4	1	2	3	0	5	6	7	8	9
S[2] = 2, P:	4	7	2	3	0	5	6	1	8	9
S[3] = 2, P:	4	7	0	3	2	5	6	1	8	9
S[4] = 0, P:	4	7	0	5	2	3	6	1	8	9
S[5] = 4, P:	4	7	0	5	2	3	6	1	8	9
S[6] = 1, P:	4	7	0	5	2	9	6	1	8	3
S[7] = 1, P:	4	7	0	5	2	9	1	6	8	3
S[8] = 1, P:	4	7	0	5	2	9	1	8	6	3
S[9] = 0, P:	4	7	0	5	2	9	1	8	3	6
P:	4	7	0	5	2	9	1	8	3	6

Figure 2.1.5: Example of conversion from a signature to a permutation

2.1.5.1 One-Point, Two-Point, and Uniform Crossover of Signatures

One-point crossover: Given strings (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm:

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) A crossover point p,$0 < p < n$, is selected uniformly, and then for all i such that $p \leq i < n$, swap $A'[i]$ and $B'[i]$

Two-point crossover: Given strings (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm:

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) Crossover points p_1 and p_2, $0 < p_1 < p_2 < n$, are selected uniformly, and then for all i such that $p_1 \leq i \leq p_2$, swap $A'[i]$ and $B'[i]$

Uniform crossover: Given strings (zero-indexed) A and B of length n, children A' and B' are created by following algorithm:

1) Assign $A' \leftarrow A$, $B' \leftarrow B$

2) For each i, $0 \leq i < n$, with a 50% probability swap $A'[i]$ and $B'[i]$

2.1.6 Hybrid Crossover (HX)

Given permutations (zero-indexed) A and B of length n, children A' and B' are created by the following algorithm:

1) Randomly select crossover method OX, CX, PMX, or MOX with equal weight.

2) A' and B' result from the randomly selected crossover method with parents A and B as input.

2.1.7 Randomizing Individuals

At times, it is poignant to measure success of genetic algorithms (as well as other problem solving techniques) against solutions derived by random search. To work within the pre-existing mechanisms of GA-specific code, random individuals were created through a contrived crossover process where for any given number of parents, their children would be an identical number of randomly generated individuals.

2.1.8 Null Crossover

Also at times, to work within the GA code, it was convenient to establish a null crossover where for any given set of parents, the resulting set of children would be identical to the set of parents. The primary purpose of null crossover was to specifically alter an individual through a different (non-crossover) process, such as mutation.

2.1.9 Greedy Approaches

Three different greedy crossover substitutes/alterations were briefly explored at points within this research.

2.1.9.1 GREEDY-1

The first greedy method randomized a population of individuals as normal, and then selected the best individual at each further iteration. This individual then underwent mutation (at a rate higher than normal) until termination of the GA. By definition of this setup, if the mutated version was the new best individual, it would then be selected for further mutations. However, if the mutation did not improve the individual, the original would remain better and be selected again for future mutations.

2.1.9.2 GREEDY-2

The second greedy method behaved like the first, however there was a chance of an *apocalypse* after each mutation that reset the population with random results. This allowed the GA to recover from overpursuit of local maxima or an otherwise non-ideal position.

2.1.9.3 GREEDY-3

The third greedy method attempted to improve the initial randomized population before employing a more traditional crossover strategy. Specifically, a random population of ten individuals was created. Then, with a standard weighted selection (described later in this chapter), an individual was selected and mutated (at a rate higher than normal). If the mutated version was superior to the original, it replaced the original individual in this version. After 200 such operations, 190 random individuals were added to the population and MOX crossover took place thereafter.

2.2 Population

Across all experiments, a steady-state population was utilized [2]. This population was kept sorted by required fitness evaluations at all times, mainly for purposes of selection and deselection. In practice, the sort routine need only be called once, upon initial population generation. Thereafter, insertions and removals, which each run in linear time, are sufficient to maintain a sorted list.

2.3 Selection

With only few exceptions (noted shortly) exactly two parents were randomly selected, weighted toward the population's better individuals. Specifically, such selection was modeled after a drawing where in a population of n individuals, the best individual has n entries, the second best has $n-1$ entries, etc., and the worst individual has 1 entry. An immediate check was put in place to prevent an individual from being selected as both parents; in such scenarios, the selection of the second parent would repeat so long as it was the same individual as the first.

For purposes of some experiments focusing on greedy approaches, the selection algorithm was simply to always choose the best individual as parent. In this research, such selection was always carried out as a single-parent operation so that truly only the best individual was ever selected for manipulation.

2.4 Deselection

In these experiments, only the worst individuals were deselected (or removed from the population) to make room for new children. That is, for a crossover method requiring n parents, their children always took the place of the worst n individuals in the population. This is also the case even in instances where children may be worse performers than the individuals they replace. In other words, children always enter the population, regardless of their quality.

2.5 Mutation

For each individual (A of length n, zero-indexed) created by means of a crossover operation, mutation was performed on it in the following manner: For each i, $0 \leq i < n$, there exists a 1% probability of randomly swapping $A[i]$ with $A[j]$, where j is a random value such that $0 \leq j < n$.

In previous related research, mutation rate was analyzed with the PMX crossover method. Figure 2.5.1 and Figure 2.5.2 show the fitness evaluations required to solve the 500-queens problem over a variety of mutation rates. Values along the y-axis depict percentile performance of 101 runs of the GA at given mutation rate. Given this data, 1% was then selected as a mutation rate for all further experiments in this study.

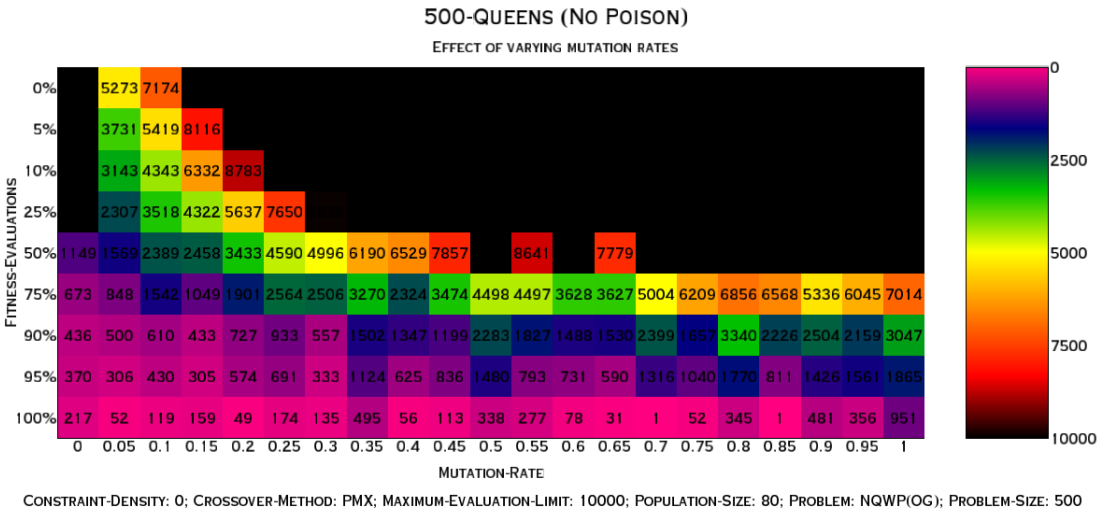

Figure 2.5.1: NQ-500: Variation of mutation rates with PMX crossover

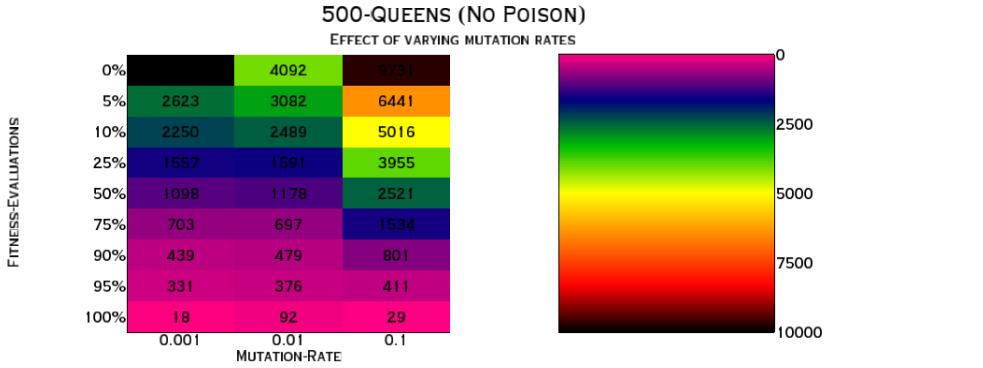

Figure 2.5.2: NQ-500: Variation of mutation rates with PMX crossover; additional granularity

Chapter 3

N-Queens With Poison (NQWP)

In the game of chess, the queen is a piece that may attack opponent pieces that share the same row, column, or one of the two diagonals as the queen. The goal of the n-queens (NQ) problem is to find one or more arrangements of n queens on an $n \times n$ chessboard such that no two queens are in position to attack each other [1]. For example, Figure 3.0.1 depicts one possible solution to the 8-queens problem.

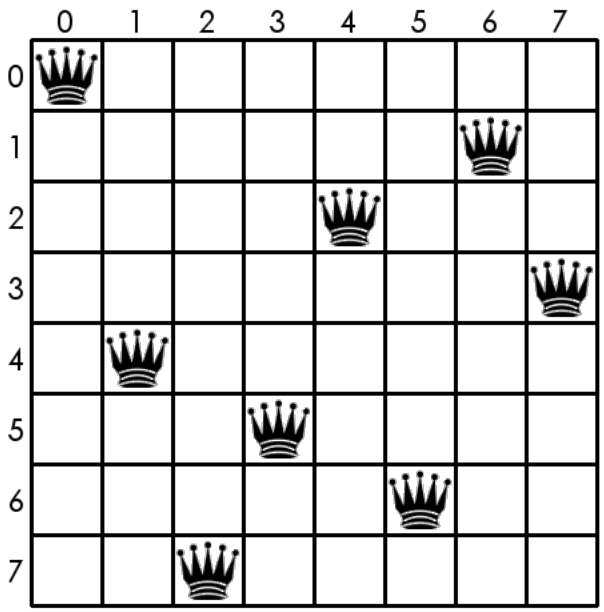

Figure 3.0.1: A solution to the 8-queens problem

In the n-queens with poison problem, by definition, squares may be deemed as poison and queens may not be placed on such squares. In general, any number or configuration of squares may be poisoned. Figure 3.0.2 depicts an 8×8 board with 10 poisoned squares, and Figure 3.0.3 shows how the solution contained in Figure 3.0.1 does not conform to the particular poisoned layout from Figure 3.0.2.

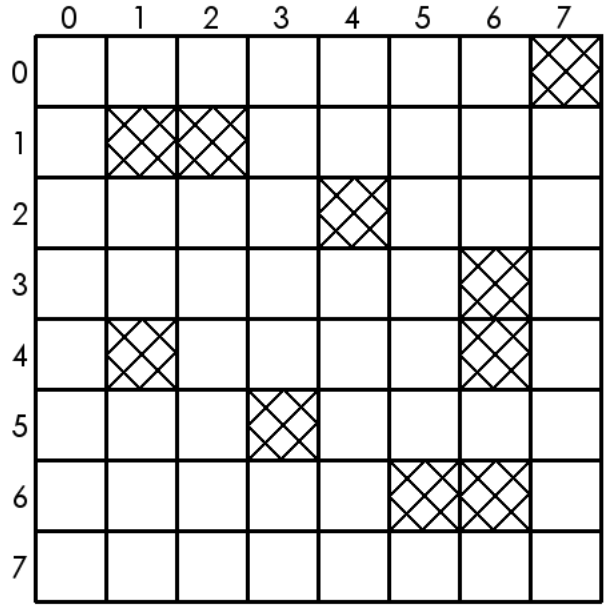

Figure 3.0.2: An 8×8 board with 10 poisoned squares

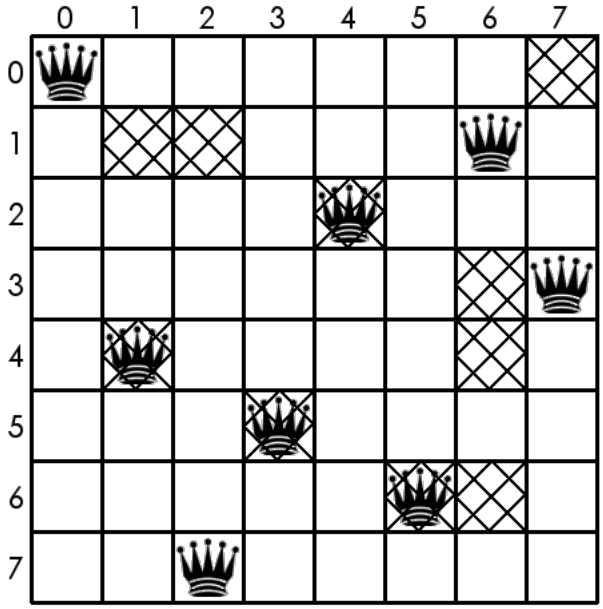

Figure 3.0.3: Solution from Figure 3.0.1 overlaid onto poisoned board from Figure 3.0.2

It is clear from the example that NQWP potentially places considerable constraints upon the NQ problem. The addition of poisoned squares has potential to make solutions to the NQ problem unsuitable as solutions for the NQWP problem depending on the configurations of poison. Figure 3.0.4 shows a successful solution to the 8-Queens problem under the poison constraints from Figure 3.0.2.

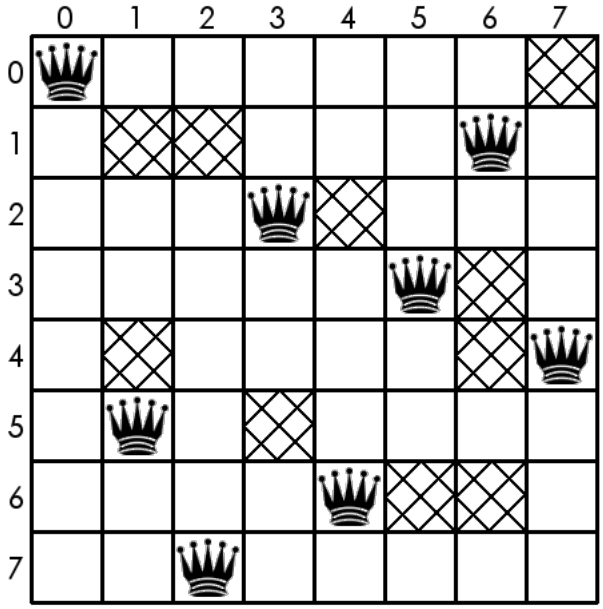

Figure 3.0.4: A solution to the 8-queens with poison problem, constrained by the board from Figure 3.0.2

3.1 Difficulty

Difficulty is measured by considering results after altering the percentage of poisoned squares across multiple trials of the NQWP problem for various values of n.

3.2 Problem Representation

A solution to the NQWP problem is represented by a permutation P of length n. Specifically, in the simplest representation, i is the column in which a queen is to be placed, and $P[i]$ contains the row. For example, the representation of the queen layout from Figure 3.0.4 would be [0 5 7 2 6 3 1 4]. However, for ordered greed fitness evaluation purposes, a permutation represents an ordering of rows. For each row, a queen is placed as far left as possible each time by avoiding poisoned spaces

as well as spaces that are susceptible to attack by previously laid queens. Fitness is measured as the percentage of queens (out of n) that can be placed legally under the leftmost-placement guideline until a single placement fails [1].

3.3 Problem Set Creation

Instances of the NQWP problem were created through the following process:

1) Choose a constraint density c such that $0 \leq c \leq 100$ (that is, the percentage of spaces to be poisoned, where 0% implies no spaces are poisoned and 100% means all spaces unoccupied by queens are poisoned)

2) Choose number of queens n

3) Using a GA, solve an instance of NQWP on an $n \times n$ board with 0% constraint density

4) Poison c% of the unoccupied spaces

5) Remove queens from the board

3.4 Solution Approach

Instances of the NQWP problem were solved by the following algorithm:

1) Begin with a board instance B created by the process described in Subsection 3.3

2) Choose a crossover method X from those described in Section Section 2.1

3) Select the number of times r to run the GA on board B with crossover method X

4) Using a GA, solve for board B utilizing crossover method X a total of r times, storing the fitness evaluations from each run in array V

5) Sort V, best to worst, and output the values at the following percentiles as array W: 100, 95, 90, 75, 50, 25, 10, 5, 0

3.5 Results

Especially given the breadth of data accumulated, there are several meaningful ways in which one could interpret GA output data, especially in terms of assigning overall winners and losers. To focus results, while the entirety of data will be considered at times, most analysis will involve discussion of *representative data*. In this thesis, for a given set of data performed upon identical parameters, an experiment was performed that ran the GA 101 times, storing sorted results at a range of percentiles from 100% (best) to 0% (worst). Additionally, such experiments upon identical parameters were also performed several times, the only effective difference being the actual random configuration of constraints, albeit still at the same constraint density. Such sets of data performed at identical configurations were then sorted by the 50th percentile (median) values, using 75th percentiles as tie-breakers, and successively utilizing higher percentiles as further tie-breakers as necessary. After this sort, the median data set was selected as the representative data for a given set of parameters. In cases where an even number of data sets were taken for a particular configuration of parameters, the better of the two middle data sets was selected as the representative data. While this is not a median in the truest sense, it is sufficient to analyze trends in data. Most of the time, the representative data was selected as the median of seven sets. However, given the lengthy running time of experiments, data was collected from a lesser number of tests (specifically, five or three) for some entire experiments.

As can be seen in later analysis, it is important to note that such representative cannot be assumed to be representative of qualities other than what it was selected as the median of medians. For example, the representative data selected could have had the worst or near-worst performance at the 95th percentile or the best or near-best performance at the 25th percentile. This could further complicate potential discussion of overall winners and losers. However, by looking at medians, the data is hopefully least skewed by random factors such as unusually easy or difficult problem sets.

It is important to preface the analysis of results by noting that there in general is not an absolute clear-cut winner as far as choosing crossover methods in conjunction with population sizes. Differences in results were often slight and very few overarching generalities could be claimed.

In the dialogue within this chapter, the term NQWP-x refers to the NQWP problem at problem size x, that is, on an $x \times x$ board. Other more fundamental alterations of GA parameters will also be reflected in the problem name. NQWP-32 was utilized as the baseline data set.

Of special note, it follows that when a GA terminates with a value less than the population size, it will have found the ideal solution randomly without ever applying any genetic crossover; this is common for easier problems. It also must be noted that after running many tests utilizing signature crossover, a bug was discovered that did not allow the GA to terminate when finding the ideal solution in this initial population generation stage; the GA could not terminate until at least 1 reproduction had taken place. This was corrected upon its detection, however many GA experiments reflect this earlier error. While this may stand out visually at times, it is important to remember that so long as the GA *actually needed* at least one reproduction to reach an ideal solution, this condition is irrelevant; it only appears in data depicting the

easiest problems: ones solvable within the initial population stage. Therefore, it was not necessary fix or redo such tests since it had no bearing on non-trivial data sets.

3.5.1 Difficulty

3.5.1.1 Primary Results

One of the few overarching statements that can be made, NQWP-32 was most difficult at a constraint density of 0.8, with difficulty tapering off at constraint densities both higher and lower than that mark. Consider Figures 3.5.1.1 through 3.5.1.1 that show 50th percentile results across all crossover methods as support.

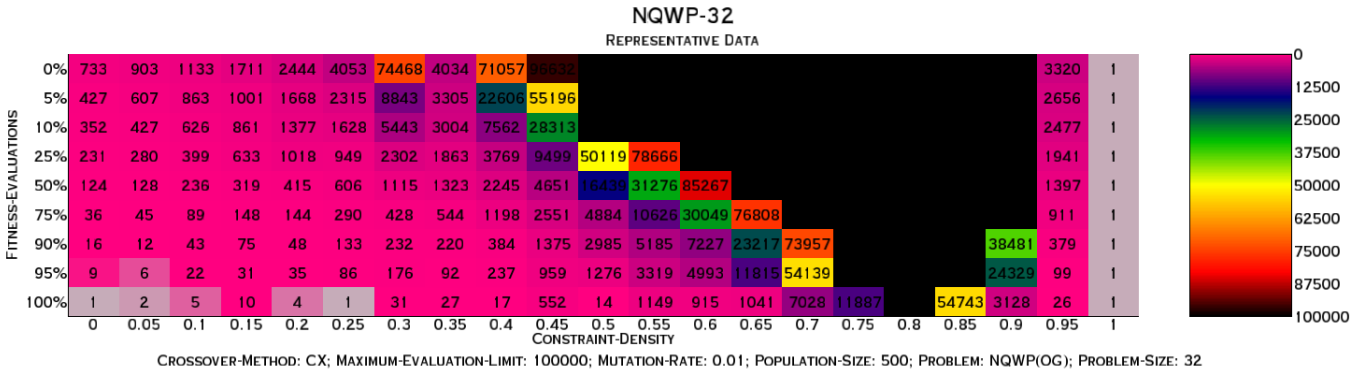

Figure 3.5.1: CX crossover at population size 500

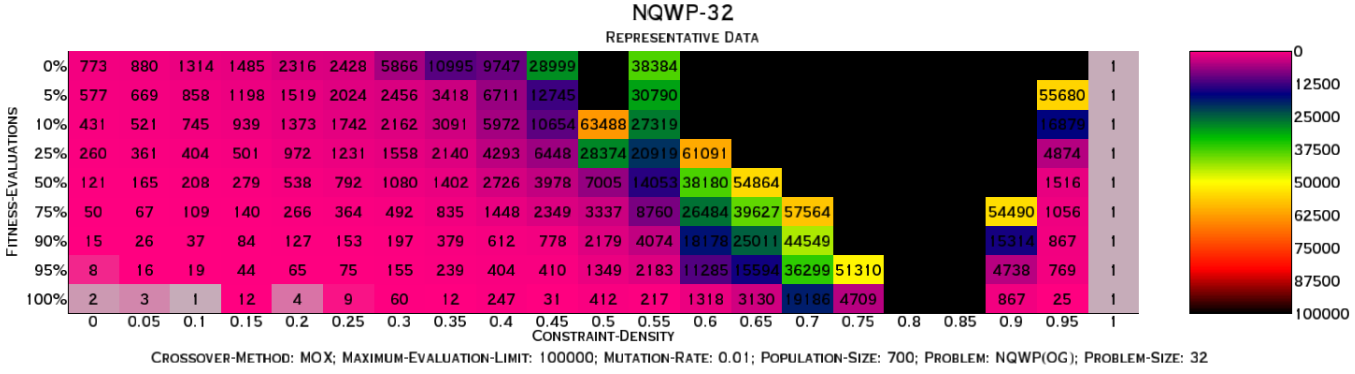

Figure 3.5.2: MOX crossover at population size 700

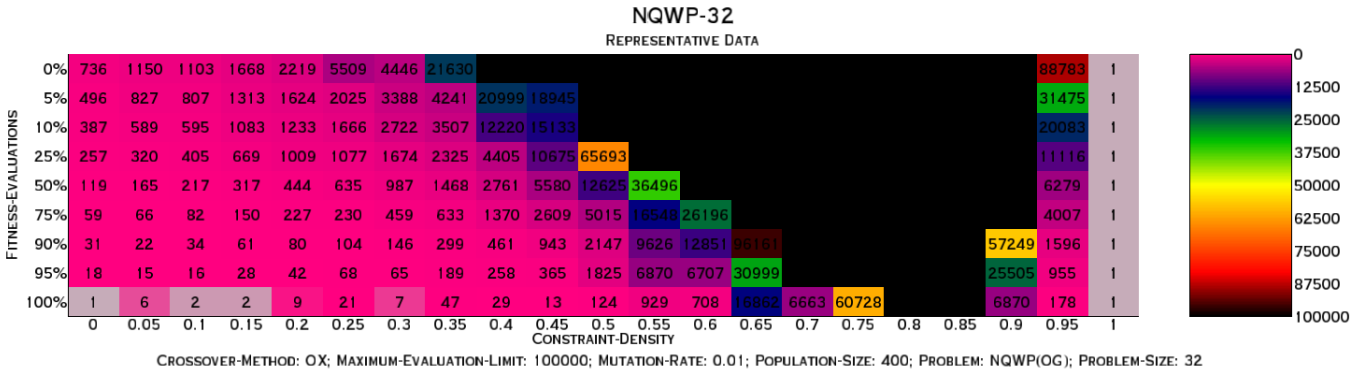

Figure 3.5.3: OX crossover at population size 500

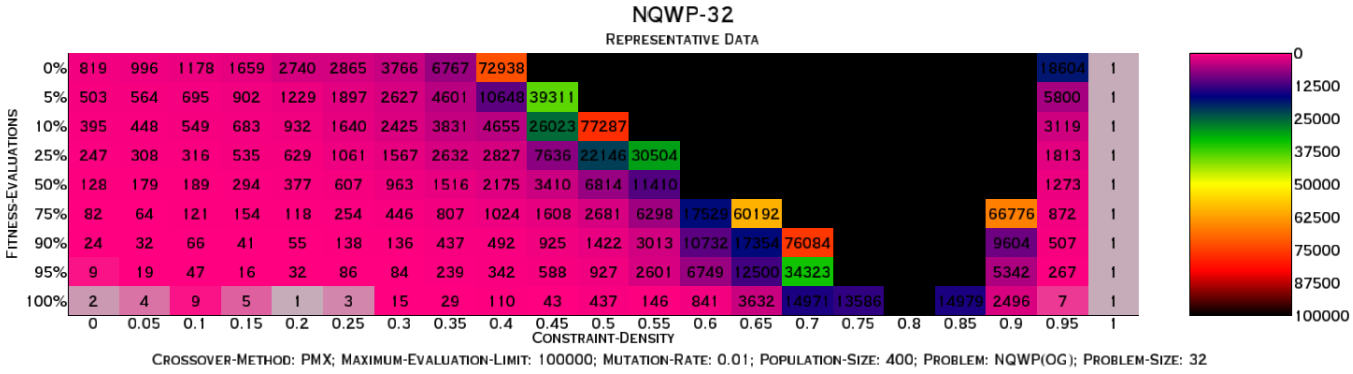

Figure 3.5.4: PMX crossover at population size 400

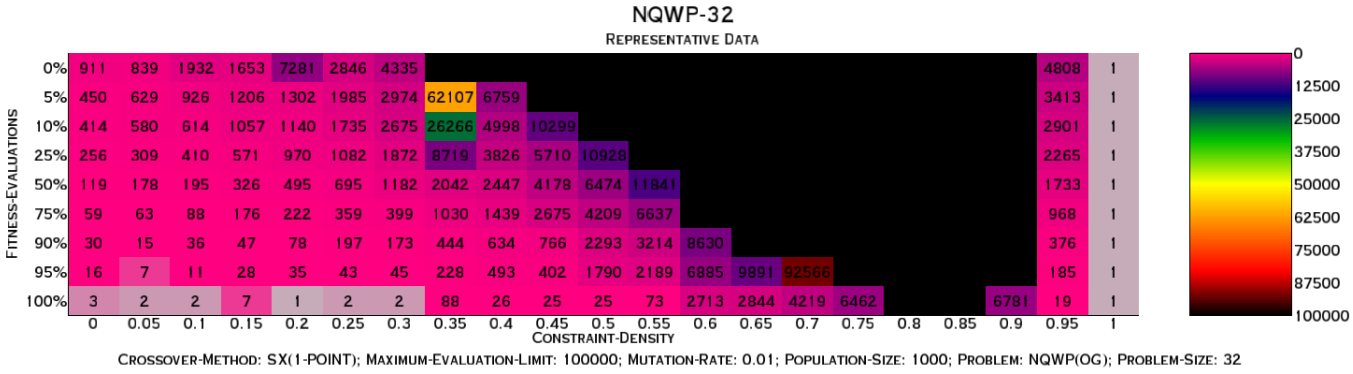

Figure 3.5.5: SX-1POINT crossover at population size 1000

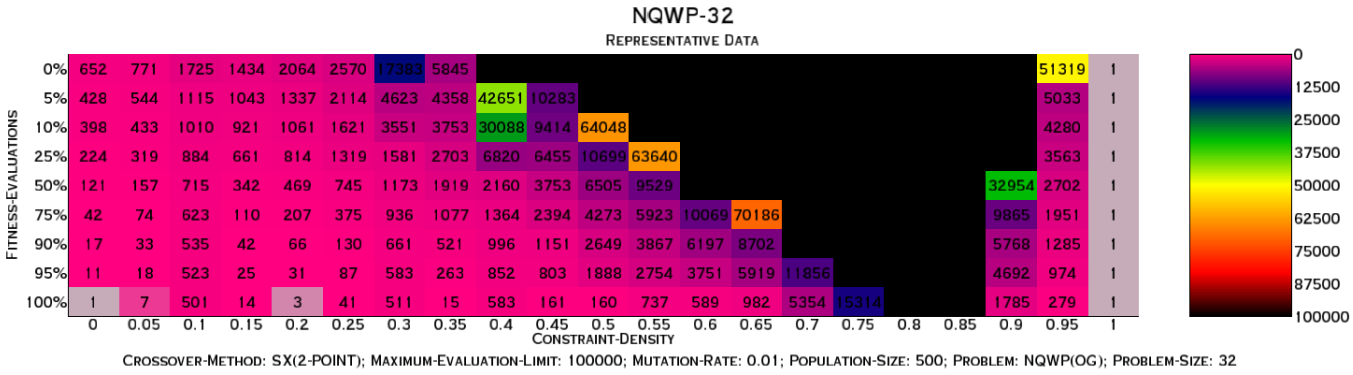

Figure 3.5.6: SX-2POINT crossover at population size 500

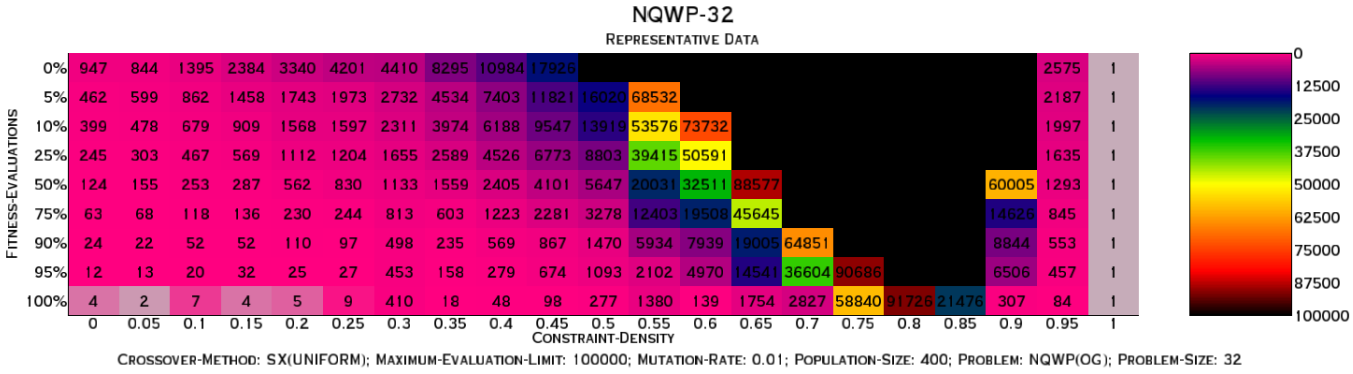

Figure 3.5.7: SX-UNIFORM crossover at population size 400

This is also supported by looking at random solutions to this problem. Figure 3.5.1.1 similarly shows problem difficulty centering around a constraint density of 0.8.

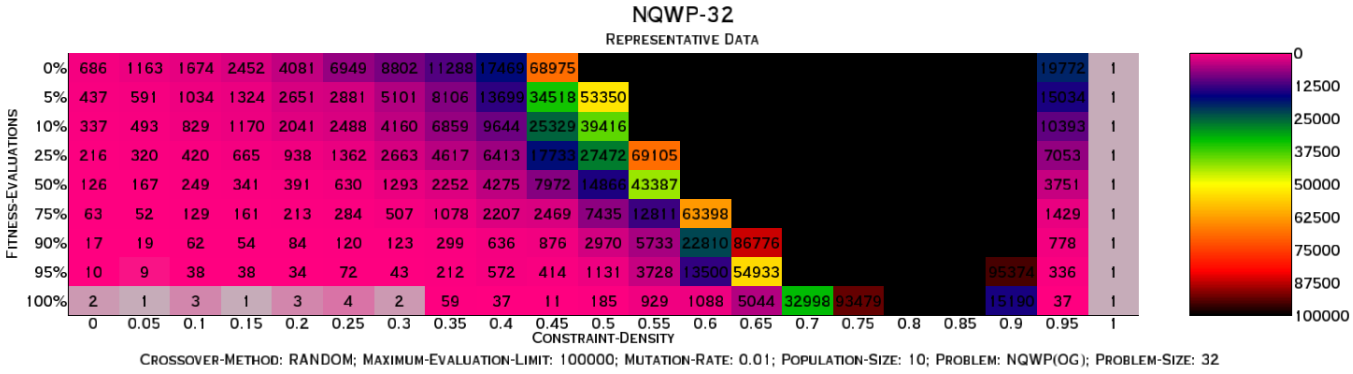

Figure 3.5.8: RANDOM search

Additionally, even at the 95th percentile of results, no crossover exhibited any solutions within 100,000 fitness evaluations for a constraint density of 0.8.

3.5.1.2 Effect of Crossover Technique on Difficulty

As can be seen in Figures 3.5.1.1 through 3.5.1.1, choice of crossover technique had no discernable effect on problem difficulty, as problems at or near constraint densities of 0.8 were always the most difficult.

Within some individual data sets, there were occasionally instances in the lower range of constraint densities where a less constrained problem required more fitness evaluations to solve than a more constrained one. While it is possible that there may be a more fundamental underlying rationale, it is believed that it is more likely that data sets involved in such comparisons contained a higher than normal number of unusually easy or difficult data sets.

3.5.1.3 Effect of Population Size on Difficulty

Similarly, population size also had no discernable effect on where NQWP is difficult. While the larger problems were certainly more difficult in the most general sense, the range of difficulty in terms of constraint density did not change. This can be evidenced by Figures 3.5.1.3 through 3.5.1.3 that depict fitness evaluations required to solve the problem for various population sizes and the standard range of constraint densities.

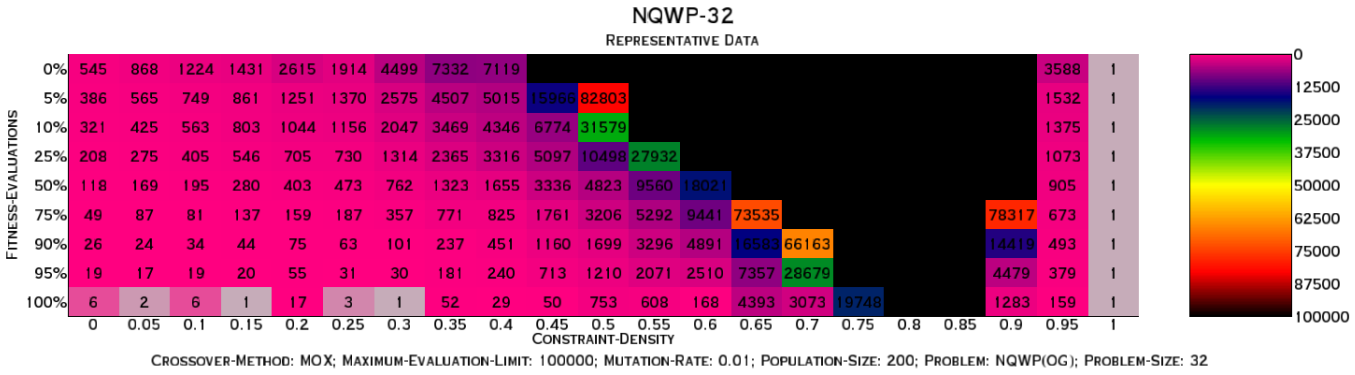

Figure 3.5.9: NQWP-32: MOX crossover at population size 200

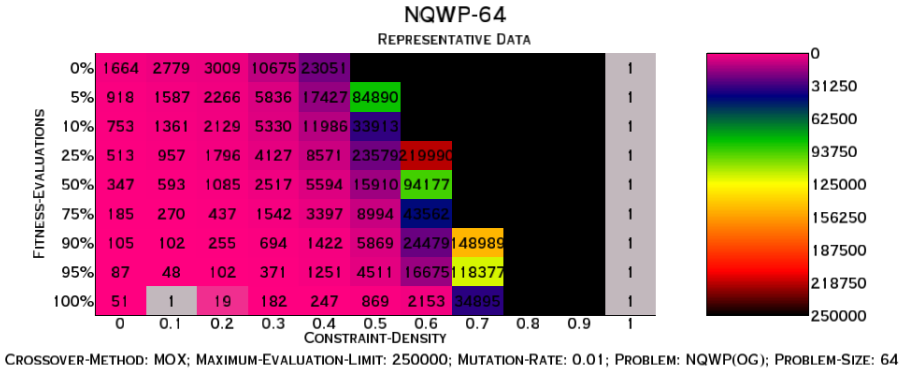

Figure 3.5.10: NQWP-64: MOX crossover at population size 200

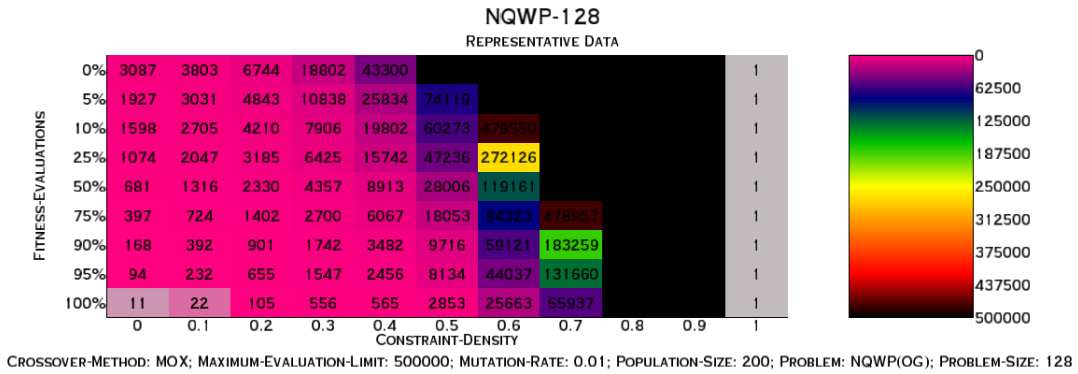

Figure 3.5.11: NQWP-128: MOX crossover at population size 200

3.5.1.4 Distribution of Difficult Problems

Altogether, there were 2099 pieces of representative data from unique combinations of GA configurations for NQWP-32. At the 25th percentile mark, 803 of them terminated only at the 100,000 fitness evaluation limit. (This general scenario shall be called limit termination.) 145 of these had population sizes of 10, which in general did not prove to yield desirable results. For other population sizes, constraint densities from 0.7 to 0.9 were represented in this 25th percentile limit-termination group. At the 5th percentile mark, and again throwing out results from population size 10, there exist limit-terminated problems at constraint densities ranging from 0.45 to 0.9.

Across all data (not just representative data) and after removing results achieved with a population size of 10, at the 50th percentile mark, limit-terminated runs can be found at constraint densities of 0.5 to 0.95. At the 5th percentile mark, limit-terminated runs begin to appear at a constraint density of 0.4. At the 0th percentile mark (showcasing the worst results), limit-terminated runs appear as early as at constraint densities of 0.1, and then appear more frequently starting at 0.3.

In summation, difficult problems do appear at constraint densities outside of those close to or equal to 0.8, however there are predictably less at constraint densities furthest away.

3.5.1.5 GA Behavior for Difficult Problems

As can be seen from the data highlighted in Subsections 3.5.1.1 on page 35 and 3.5.1.3 on page 38, at the moderately difficult constraint density ranges centering roughly around 0.6, there are data sets that are limit-terminated some or most of the time, however can still be solved in relatively few fitness evaluations in the better-case scenarios. These cases exhibit the important effect that quality initial random conditions

may have on the ability of the GA to solve the problem.

3.5.1.6 Characteristics of Difficult Problems

Across problem sets created with identical constraint density, less poison in the left-most columns (or more generally, the left half of the board) and then consequently having more poison in the right portion of the board is a major factor in a data set having high difficulty. As the OG algorithm fills columns in as left-to-right as possible, the left columns comparatively get filled before their counterparts on the right. It should then come as no major surprise that with more poison (fewer options to place a queen) toward the end of the algorithm, the algorithm is more likely to fail having to work in this exceptionally constrained area of data.

Consider Table 3.5.1 that depicts this phenomenon for five pairs of data sets. The first four were selected for exhibiting vast differences in difficulty. The fifth was selected for exhibiting roughly similar performance. It is clear that in each of the first four columns, the values peak around the half-way point of the list, indicating substantially more open space on the left half than on the right.

Constr. Dens.	i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Data set A: 0.5		-1	-1	-3	2	7	7	9	9	10	9	10	13	16	15	24	20	28	29	30	24	21
Data set B: 0.8		2	0	2	-3	-4	-2	-2	0	3	-2	2	6	10	16	19	22	17	21	23	21	20
Data set C: 0.55		-1	-1	8	15	19	18	16	13	18	19	20	20	17	19	21	22	22	22	20	16	15
Data set D: 0.5		0	-3	2	3	10	13	3	0	3	8	4	3	8	8	15	14	19	16	15	17	15
Data set E: 0.4		3	-1	-2	0	0	-1	-4	-6	-10	-5	-4	-7	-5	-3	8	7	6	4	6	0	-1

Table 3.5.1: NQWP-32; Difference between total number of non-poisoned squares in columns 0 through i for $i \leq 20$

3.5.2 Crossover Techniques

3.5.2.1 Primary Results

Overall, results tend to show that MOX, SX-2POINT, SX-UNIFORM, and PMX stand out as the best performers, perhaps in that order, although certainly not beyond argument.

For constraint density 0.25, at which problems are easy but not quite trivial (the problem is not generally solved within the population initialization stage), performance differences between any configuration are slight. SX-UNIFORM has the best result overall, found at population size 20, and CX has the worst individual best result, found at population size 100. Every other crossover method shows best results at population size 50. Overall, the GA has the best results at smaller population sizes (though certainly does not perform poorly at larger ones). Figure 3.5.12 shows general performance of the GA for given combinations of crossover technique and population size; results are 50th percentile values of fitness evaluations required to solve NQWP-32 with a constraint density of 0.25.

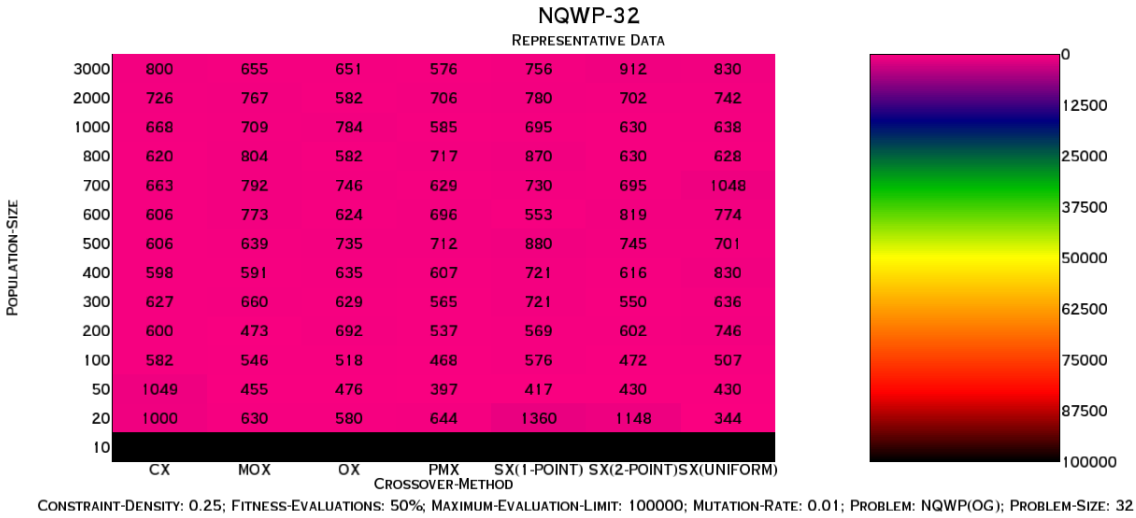

Figure 3.5.12: NQWP-32: All crossovers and population sizes for 50th percentile runs at constraint density 0.25

At a constraint density of 0.5, where NQWP certainly starts to pick up more difficulty, SX-1POINT has the overall best performance at population size 400. The worst individual best can be found with OX at population size 200. These results are found in Figure 3.5.13, which also shows that midrange population sizes are in general the best performers.

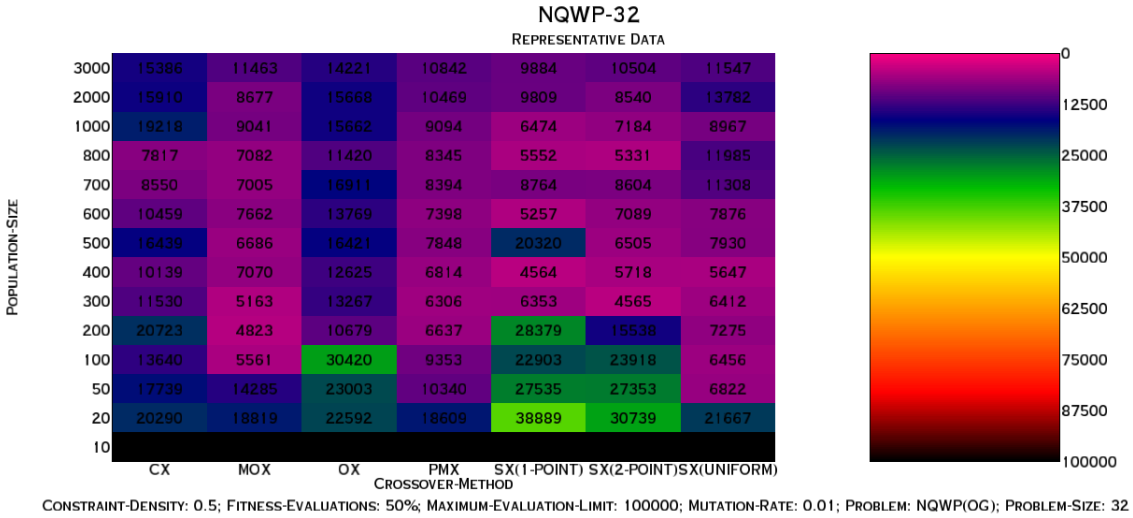

Figure 3.5.13: NQWP-32: All crossovers and population sizes for 50th percentile runs at constraint density 0.5

At an even more difficult set of problems, depicted in Figure 3.5.14, where the constraint density is 0.6, results start to show more of a divergence in quality as far as population sizes go. While MOX is the top-performing crossover technique at a population size of 200, more crossover techniques tend to find their best performances at both high and low population sizes. CX's best is at population size 20, much different than its best for constraint density 0.5, which occurred at 800.

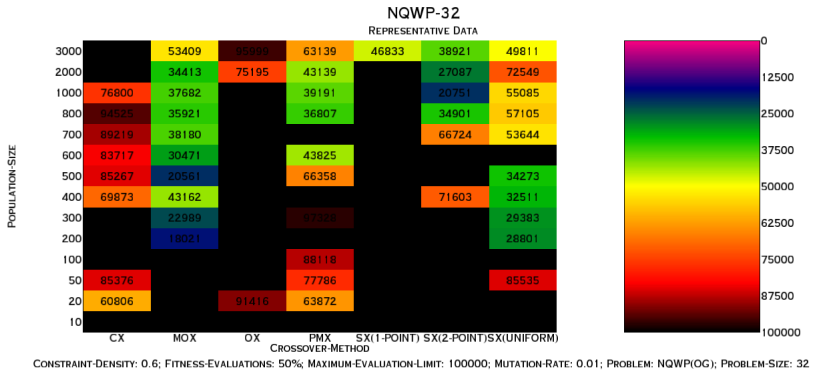

Figure 3.5.14: NQWP-32: All crossovers and population sizes for 50th percentile runs at constraint density 0.6

In consideration of 75th percentile results at a slightly tougher problem at constraint density 0.65, shown in Figure 3.5.15, results are similar. MOX retains the overall best result, this time at population 300, while other methods general find their best results at the extremes of population sizes.

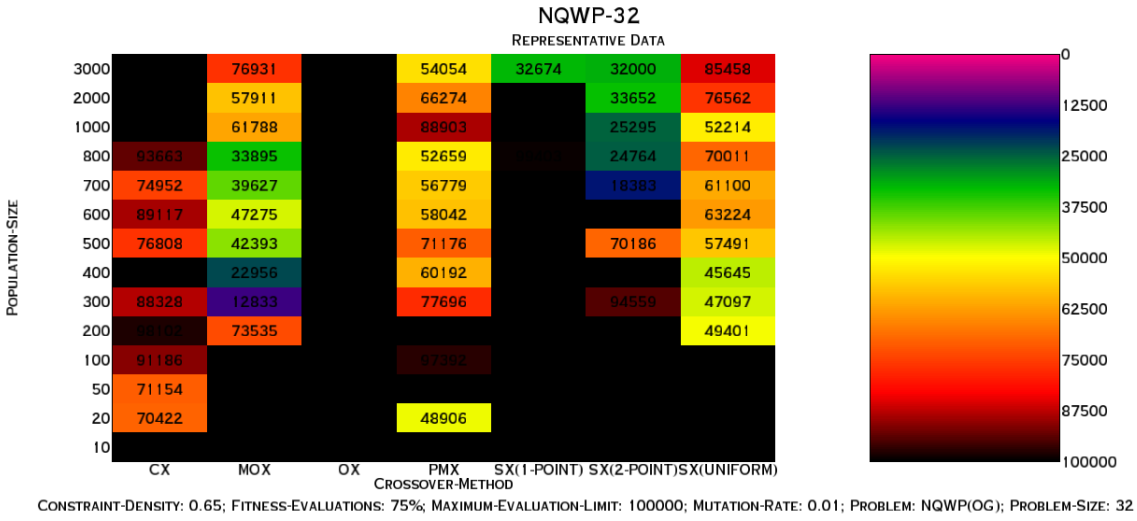

Figure 3.5.15: NQWP-32: All crossovers and population sizes for 75th percentile runs at constraint density 0.65

Additionally, consider the parameter configurations of the 20 best runs across all data (not just representative data) for various constraint densities in Table 3.5.2.1:

Constraint Density: 0.9		Constraint Density: 0.7		Constraint Density: 0.6	
SX(2-POINT)	100	MOX	700	SX(2-POINT)	700
SX(UNIFORM)	400	MOX	1000	MOX	300
SX(UNIFORM)	200	MOX	700	SX(2-POINT)	500
MOX	500	MOX	1000	MOX	200
SX(UNIFORM)	300	MOX	600	SX(2-POINT)	600
CX	500	MOX	2000	MOX	300
MOX	600	MOX	1000	MOX	200
SX(UNIFORM)	300	MOX	1000	MOX	600
MOX	600	MOX	500	MOX	300
MOX	1000	MOX	500	MOX	200
MOX	700	MOX	600	SX(2-POINT)	700
CX	700	SX(2-POINT)	2000	MOX	600
SX(1-POINT)	400	SX(2-POINT)	2000	MOX	500
MOX	700	SX(2-POINT)	700	MOX	500
SX(2-POINT)	300	SX(UNIFORM)	300	MOX	400
SX(1-POINT)	800	MOX	700	SX(UNIFORM)	200
OX	600	SX(2-POINT)	2000	SX(2-POINT)	1000
CX	400	MOX	300	SX(UNIFORM)	300
SX(2-POINT)	500	CX	20	SX(2-POINT)	3000
PMX	400	MOX	700	MOX	200
Constraint Density: 0.5		Constraint Density: 0.4		Constraint Density: 0.25	
SX(2-POINT)	200	SX(2-POINT)	100	SX(UNIFORM)	20
SX(2-POINT)	400	SX(1-POINT)	800	SX(UNIFORM)	50
SX(2-POINT)	300	PMX	50	SX(1-POINT)	50
MOX	100	SX(2-POINT)	100	SX(UNIFORM)	20
SX(1-POINT)	600	MOX	200	PMX	50
SX(1-POINT)	400	MOX	50	PMX	100
SX(UNIFORM)	400	SX(UNIFORM)	50	PMX	200
MOX	200	SX(2-POINT)	100	SX(1-POINT)	50
SX(2-POINT)	300	MOX	100	MOX	100
MOX	200	SX(1-POINT)	1000	MOX	1000
SX(1-POINT)	600	SX(2-POINT)	700	SX(UNIFORM)	20
SX(1-POINT)	400	MOX	100	SX(2-POINT)	50
MOX	600	SX(UNIFORM)	200	SX(UNIFORM)	50
SX(UNIFORM)	300	SX(2-POINT)	200	SX(UNIFORM)	20
MOX	100	SX(UNIFORM)	50	SX(2-POINT)	50
SX(UNIFORM)	100	MOX	200	MOX	50
SX(1-POINT)	1000	SX(2-POINT)	1000	OX	100
SX(UNIFORM)	200	SX(2-POINT)	600	PMX	50
SX(1-POINT)	700	MOX	100	SX(1-POINT)	100
SX(UNIFORM)	100	SX(1-POINT)	200	MOX	400

Table 3.5.2: Best 20 crossover method and population size configurations (sorted by 50th percentile values, across all GA runs) for selected constraint densities

It is again evident where MOX and SX techniques are successful overall, and PMX

stands out in easier problems.

3.5.2.2 Effect of Population Size on GA Performance

As has been alluded to in Subsections 3.5.2.1 and 3.5.1.1, performance can be quite dependant on population size. It can be seen that more moderate population sizes often attack the most difficult problems well, however both large and small population sizes have a chance to attack difficult problems well. Large population sizes especially may find solutions to difficult problems, albeit at higher numbers of fitness evaluations.

3.5.2.3 Crossover Variations and Substitutions

GREEDY-1 (Subsection 2.1.9.1 on page 24) was tested on NQWP-32 and undoubtedly failed. In all tests, only once did it not limit-terminate at 90th percentile (or better) results.

More success (but still worse than crossover: Figure 3.5.1.3 on page 39) was achieved with GREEDY-2 (Subsection 2.1.9.2 on page 24). First, it was run on NQWP-128, and it was determined that mutating an individual at a rate of 5% would be a solid parameter.

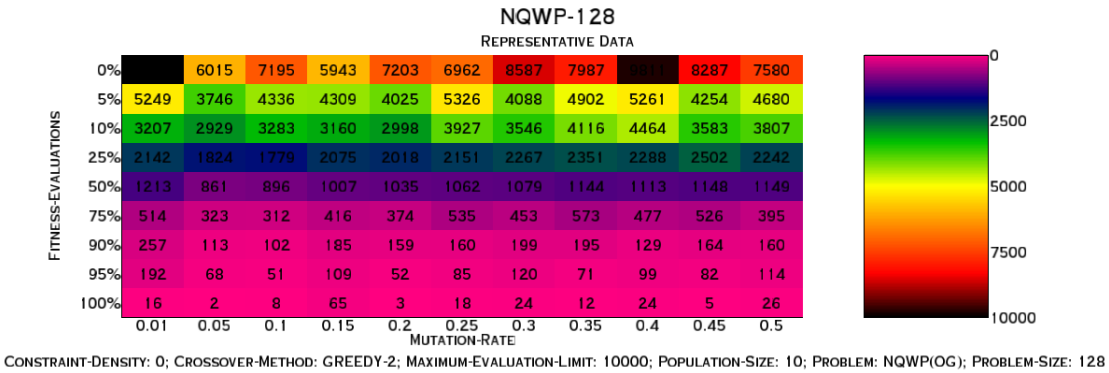

Figure 3.5.16: NQWP-128: GREEDY-2; 5% chance of apocalypse

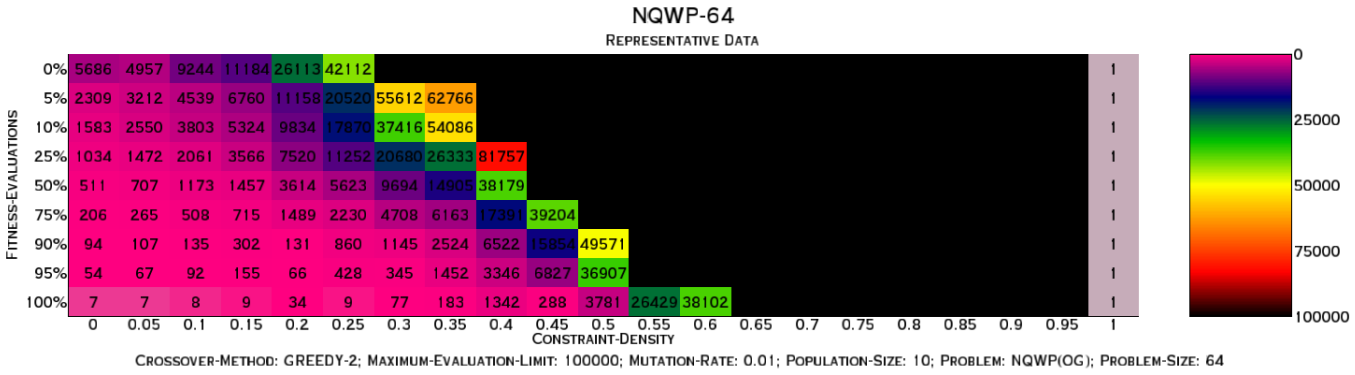

Figure 3.5.17: NQWP-64: GREEDY-2; 5% chance of apocalypse

GREEDY-3 (Subsection 2.1.9.3 on page 25) has more success, generally defeating its most analogous counterpart at the less constrained, easier problems, however MOX at population size 200 outperforms GREEDY-3 at the more difficult problems.

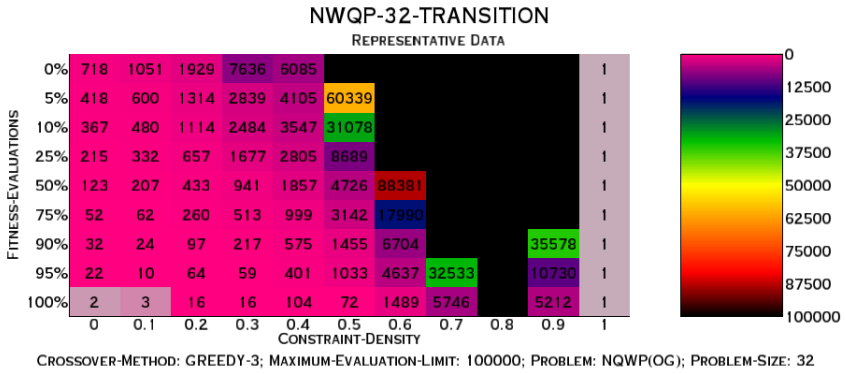

Figure 3.5.18: NQWP-32: GREEDY-3

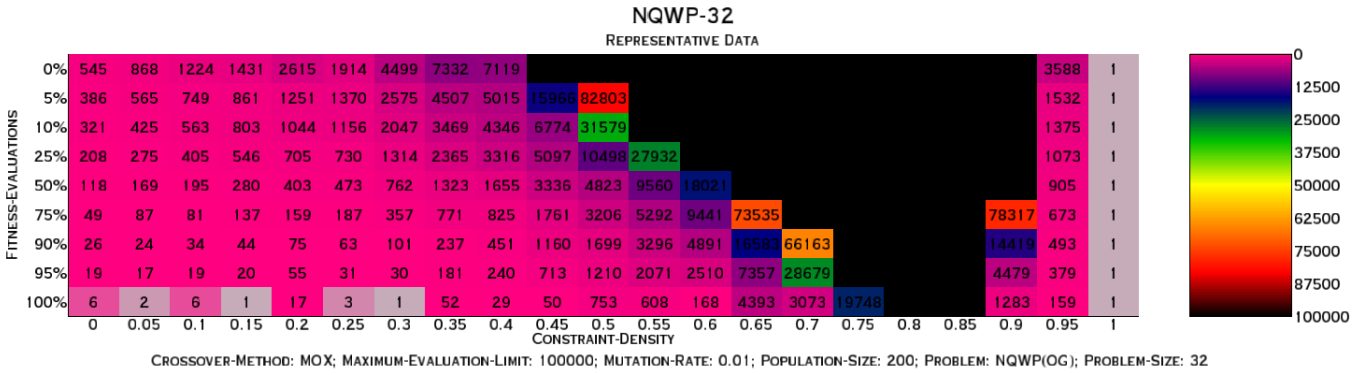

Figure 3.5.19: NQWP-32: MOX at population size 200

Chapter 4

Processor Scheduling (PS)

In an array of m processors that operate over a duration of n units of time, there are $m \cdot n$ single-time-unit processes that can be scheduled under such a configuration. In addition, precedence relationships can exist mandating that certain processes are to be completed before others. The PS problem is defined as scheduling these $m \cdot n$ processes into a perfect schedule (that completes in n time-units) under a particular set of precedence constraints [3].

4.1 Difficulty

Difficulty is measured by considering results after altering the percentage of existing precedence relations across multiple trials of the PS problem for various values of n.

4.2 Problem Representation

In the PS problem, the process schedule can be represented by a permutation P of length $m \cdot n$, where a process in P at index i can be said to begin at time i mod m and on processor i div n. For OG fitness evaluation purposes, the permutation contains an ordered list of processes where each process is placed in the first available slot that does not violate any precedence requirements [3]. Fitness is measured as the

percentage processes (out of $m \cdot n$) that can be scheduled legally under the first-available-slot guideline until a single attempt fails.

4.3 Problem Set Creation

Instances of the PS problem will be created under the following guidelines:

1) Choose a constraint density c such that $0 \leq c \leq 100$ (that is, the percentage of precedence relations to include, where 0% implies no precedence relations and 100% means that every process requires every previously scheduled process as precedence relations)

2) Choose time n, along with number of processors m; $m = 4$ was primarily utilized throughout this research

3) Create solution S to be overlaid with the precedence relations graph, where for each i, $0 \leq i < m \cdot n$, $S[i] = i$; this is automatically a valid solution as all numbered process labels are essentially arbitrary

4) Add c% of the possible precedence relations

5) Remove the processes from the schedule

4.4 Solution Approach

Instances of the PS problem will be solved by the following algorithm:

1) Begin with precedence relation set Q created by the process described in Subsection 4.3

2) Choose a crossover method X from those described in Section Section 2.1

3) Select the number of times r to run the GA on precedence relation set Q with crossover method X

4) Using a GA, solve for precedence relation set Q utilizing crossover method X a total of r times, storing the fitness evaluations from each run in array V

4a) There exists the opportunity to topologically sort individuals prior to fitness evaluation to increase their quality (and thereby ideally reducing the number of fitness evaluations necessary to reach a solution)

5) Sort V, best to worst, and output the values at the following percentiles as array W: 100, 95, 90, 75, 50, 25, 10, 5, 0

4.4.1 Topological Sorting

Three different sorts were used in analysis of the PS problem. The goal of each sort is to place processes with low or no in-degree early in the list, remove their dependencies, and then successively apply this step. As will be described in the results section later on, fully sorting the data sorted it so well that GAs could not improve upon the process. In addition to its effectiveness, the full sort is also quite time consuming. Other sorts were performed, along with an eventual analysis on their true runtimes, in hopes that one might improve upon the actual runtime of the full sort, however no overall improvements in runtime were made.

4.4.1.1 Full Sort

Algorithm 4.1 Full Topological Sort for Processor Scheduling

For precedence relation set Q and a (zero-indexed) individual A of length n, create list A' as follows:

1) Move each orphan process in A to the end of A' maintaining relative position from A (first orphan process in A appears before all other orphan processes in A', etc.); assign k to be the number of orphan processes (the orphan processes are stored in $A'[n-k-1]$ through $A'[n-1]$)

2) For each i, $0 \leq i < n-k$, find the first occurrence of a process in A with an in-degree of zero (that is, a process that does not require another process to complete before it can begin); call it p_i

3) Assign $A'[i] \leftarrow p_i$

4) Remove all precedence relations from Q that begin with p_i

5) Remove p_i from A

6) Repeat steps 2-6 until A is empty

7) (Permanently) assign $A \leftarrow A'$

4.4.1.2 Rough Sort

Algorithm 4.2 Rough Topological Sort for Processor Scheduling

For precedence relation set Q and a (zero-indexed) individual A of length n, create list A' as follows:

1) Move each orphan process in A to the end of A' maintaining relative position from A (first orphan process in A appears before all other orphan processes in A', etc.); assign k to be the number of orphan processes (the orphan processes are stored in $A'[n-k-1]$ through $A'[n-1]$)

2) Assign $i \leftarrow 0$

3) For $j = 0$, $0 \leq j < n$, find the first unmarked occurrence of a process in A with an in-degree of zero **or one**; call it p_i

4) Assign $A'[i] \leftarrow p_i$

5) Remove all precedence relations from Q that begin with p_i

6) Mark p_i in A

7) Increase j by 1

8) Repeat steps 2-7 until every element of A is marked

4.4.1.3 Partially Random Sort

Algorithm 4.3 Partially Random Sort for Processor Scheduling

For an individual A:

1) With 10% probability, return the results of the full sort algorithm described in Subsection 4.1

2) Otherwise, return A unchanged

4.5 Results

The generalities described for NQWP results in Section 3.5 on page 33 are applicable here as well to preface data.

Additionally, unless stated otherwise, the rough sort was utilized as the baseline sort for PS data.

4.5.1 Difficulty

4.5.1.1 Primary Results

The most difficult problems are found near (though not necessarily exactly at) constraint densities of 0.2. Consider Figures 4.5.1.1 on the next page through 4.5.1.1 on page 56, three pairs of figures that highlight general performance across all constraint densities. For each pair, the first figure shows performance at the 50th percentile whereas the second figure shows performance at the 10th percentile.

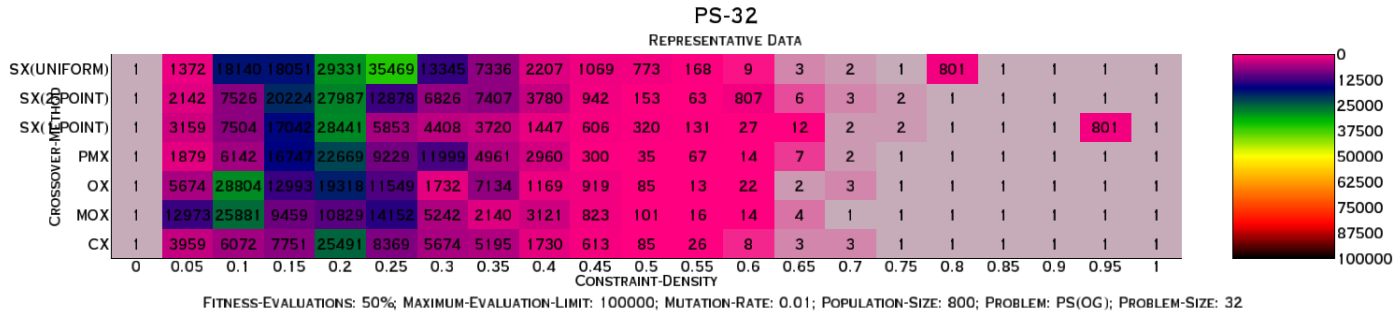

Figure 4.5.1: PS-32: All crossovers at population size 800; 50th percentile data

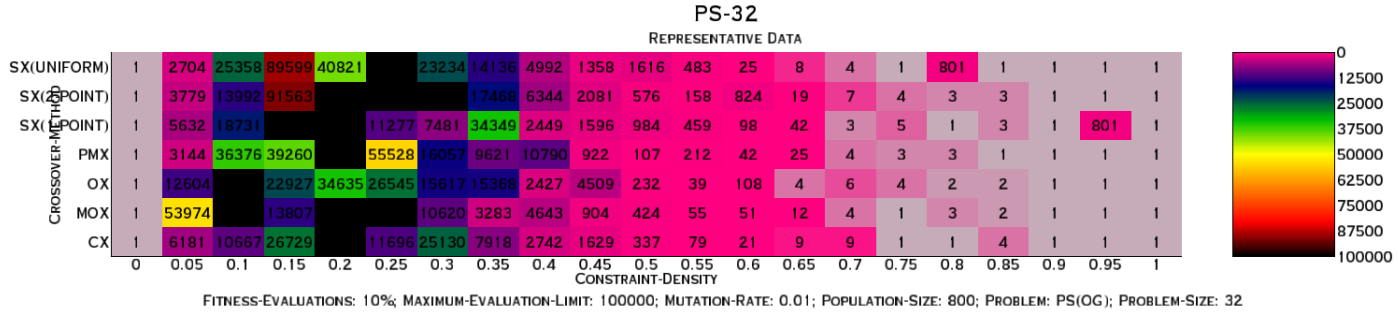

Figure 4.5.2: PS-32: All crossovers at population size 800; 10th percentile data

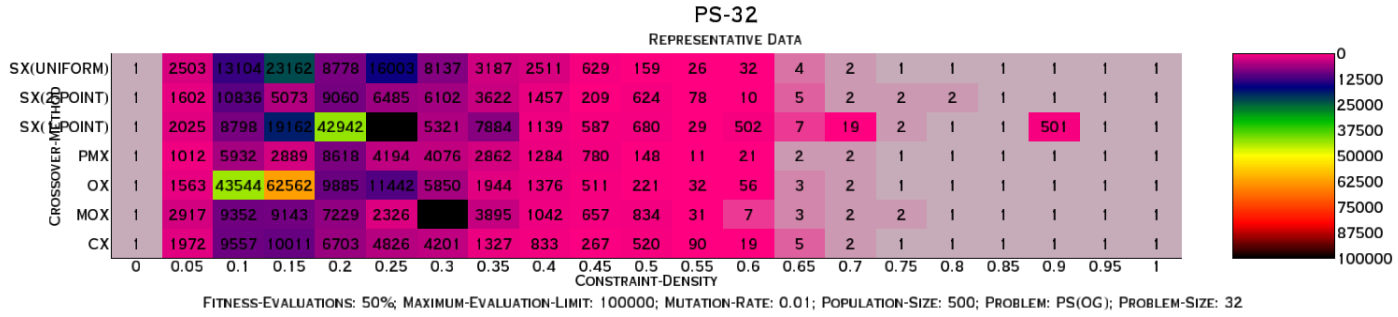

Figure 4.5.3: PS-32: All crossovers at population size 500; 50th percentile data

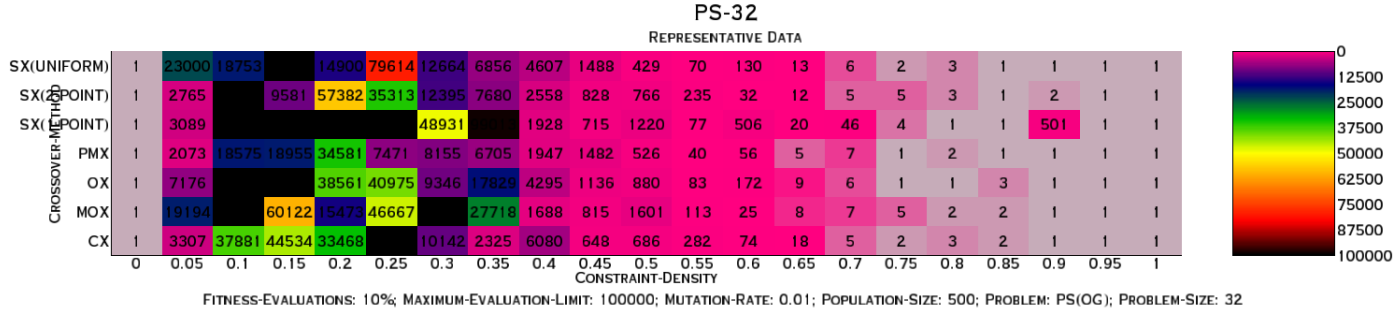

Figure 4.5.4: PS-32: All crossovers at population size 500; 10th percentile data

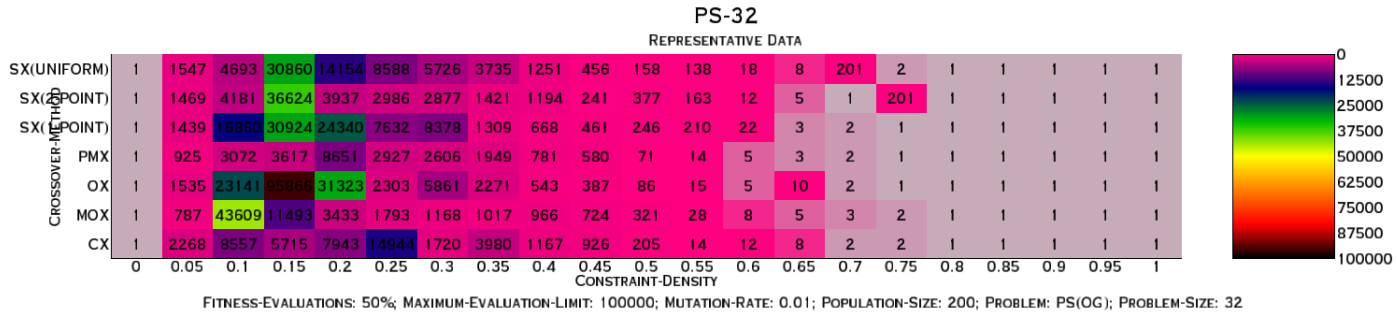

Figure 4.5.5: PS-32: All crossovers at population size 200; 50th percentile data

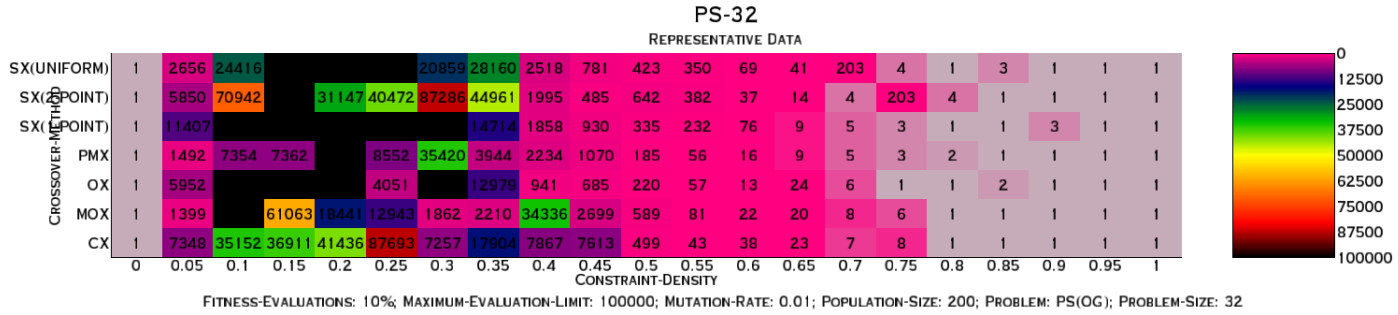

Figure 4.5.6: PS-32: All crossovers at population size 200; 10th percentile data

Additionally, Table 4.5.1 on the next page depicts the number of limit-terminations at the 50th percentile found at each constraint density, from all runs of the GA (not just representative data). Here, constraint densities of 0.2 and 0.25 stand out as the most frequently limit-terminated. At the 25th percentile (not depicted) there are 174 limit terminations for constraint density 0.2, and 144 for constraint density 0.25,

lending more credence to suggesting 0.2 is overall, perhaps the more difficult problem constraint density in general.

Constraint Density::	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.55	0.6
Limit-Terminations	1	34	60	81	95	71	34	28	10	5	0	3

Table 4.5.1: Number of limit-terminations at 50th percentile results per constraint density across all PS-32 data

4.5.1.2 Effect of Crossover Technique on Difficulty

As can be seen in the previous figures, choice of crossover technique had a mild, but noticeable effect on problem difficulty, though problems at or near constraint densities of 0.2 were still generally the most difficult. Table 4.5.1.2 shows that only with MOX was the most difficult constraint density exactly 0.2. Each other crossover method exhibited most difficult behavior (at least in terms of frequency of limit-terminated run) at 0.15 or 0.25.

Const. Density:	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.55	0.6
CX	0	0	2	3	8	5	3	1	2	2	0	0
MOX	1	5	9	21	16	15	4	6	2	1	0	1
OX	0	11	16	14	8	6	3	2	0	0	0	0
PMX	0	0	4	9	12	5	5	2	2	1	0	0
SX-1POINT	0	4	13	12	11	11	9	4	2	0	0	1
SX-2POINT	0	2	6	11	16	13	9	5	1	0	0	0
SX-UNIFORM	0	2	5	4	12	11	2	1	2	1	0	1
RANDOM	0	5	6	3	3	2	3	0	0	0	0	0

Table 4.5.2: PS-32: Number of limit-terminated runs at 50th percentile across all data

4.5.1.3 Effect of Problem Size on Difficulty

With the representative data chosen (MOX at population size 200), problem size made no discernable difference on where the PS problem was difficult, as is seen in Figures 4.5.1.3 on the next page and 4.5.1.3 on the following page in addition to previously discussed results..

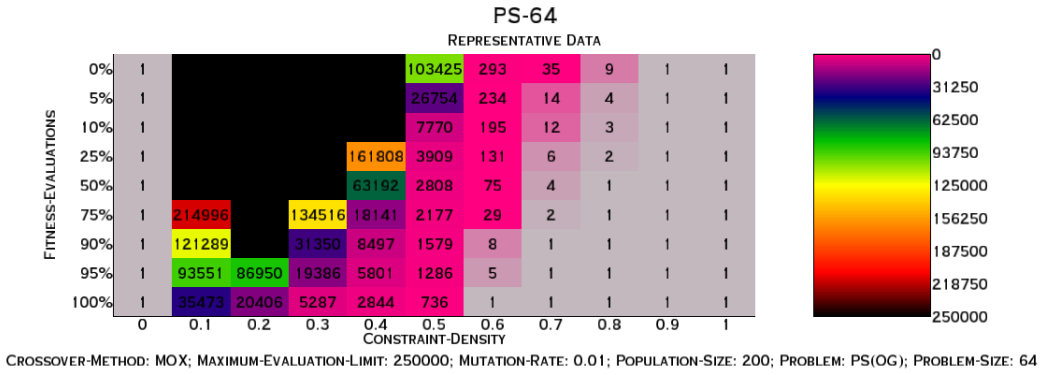

Figure 4.5.7: PS-64: MOX at population size 200

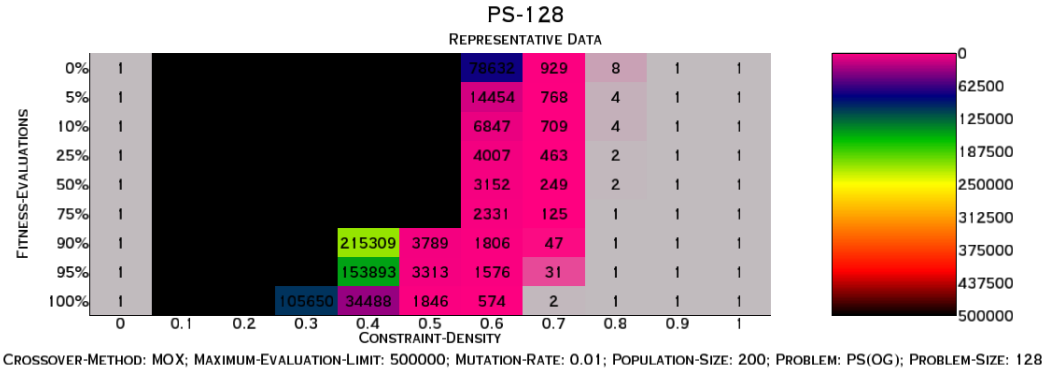

Figure 4.5.8: PS-128: MOX at population size 200

4.5.1.4 Distribution of Difficult Problems

As can bee seen in Table 4.5.1.2 on the previous page, difficult problems appear across a wide range of constraint densities, from 0.05 to 0.6.

4.5.1.5 GA Behavior for Difficult Problems

As can be seen from the data highlighted in Figures 4.5.1.1 on page 54 and 4.5.1.3 on the previous page, at the moderately difficult constraint density ranges centering roughly around 0.4, there are data sets that become limit-terminated some or most of the time, however can still be solved in relatively few fitness evaluations in the

better-case scenarios. These cases exhibit the important effect that quality initial random conditions may have on the ability of the GA to solve the problem.

4.5.1.6 Effect of Sorting Algorithm on Difficulty

Different sorting algorithms do have mild effects on what constrait densities are most difficult. With CX at population size 50, fully sorted, the most difficult problems are at constraint density 0.35, as seen in Figure 4.5.1.6. While there is also great difficulty at 0.35 for the rough sort implementation, depicted in Figure 4.5.1.6, there is also roughly equal difficulty at lower constraint densities.

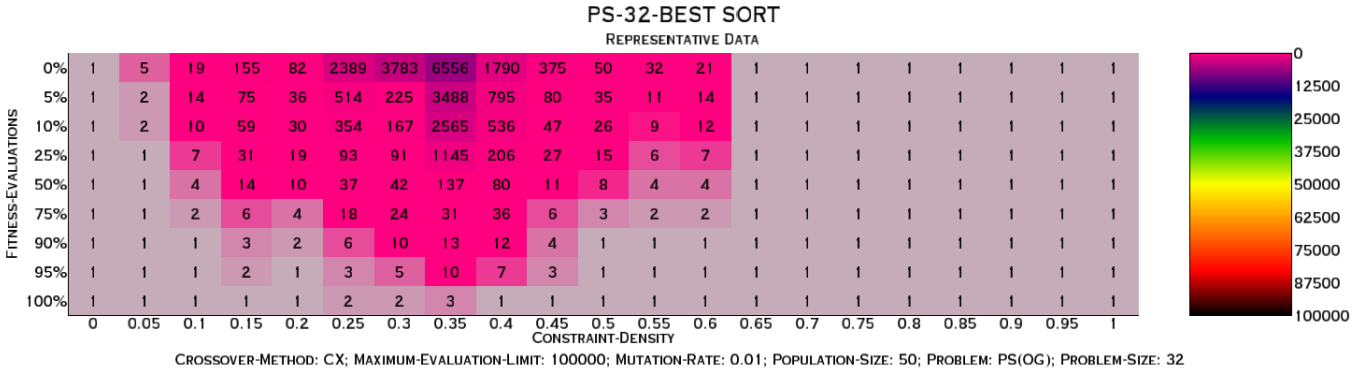

Figure 4.5.9: PS-32: Full sort, CX at population size 50

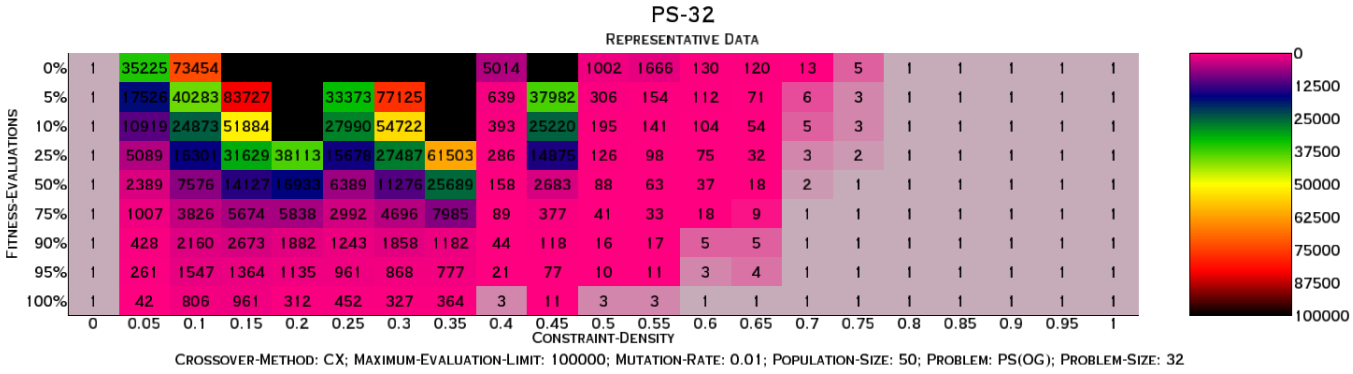

Figure 4.5.10: PS-32: Rough sort, CX at population size 50

For MOX at population size 200, the fully sorted implementation (Figure 4.5.1.6)

finds the most difficult problems at constraint density 0.3, while the roughly sorted counterpart (Figure 4.5.1.6) has the most difficulty at 0.1.

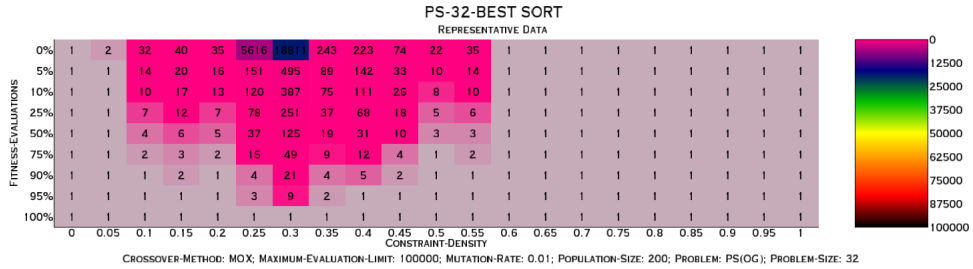

Figure 4.5.11: PS-32: Full sort, MOX at population size 200

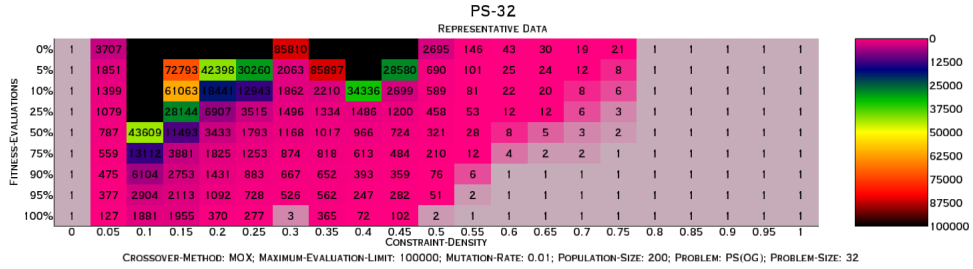

Figure 4.5.12: PS-32: Rough sort, MOX at population size 200

As sorting is a major variation of fitness calculation, it should be no major surprise that altering the sort may change where problems are the most difficult.

4.5.1.7 Effect of Processor Count on Difficulty

While a processor count of 4 was used throughout this research, a later analysis was done exploring the effects of utilizing other processor counts. Such experiments were done using RANDOM with the full sort.

The results of these tests, in Figures 4.5.1.7 on the following page through 4.5.1.7, show staggeringly different problem difficulties. The data shows that for small processor counts, the difficult problems appear at mid-range constraint densities. These are

still comparatively easy to solve, and GA performance is, overall, solid. As the processor count increases, several things occur: 1) the range of difficulty shifts to lower constraint densities, 2) the difficulty of these problems increases, and 3) the range at which difficult problems are seen decreases. (The only observed exception to this is PS-128 at 4 processors, a value which gives relatively easy problems. Coincidentally, this again is the the processor count at which all other research into this problem utilized.) This trend holds until the processor count is equal to half the problem size, at which point the problem is trivially easy.

Further analysis of such problem difficulty is left for future research. (See Section 5.2 on page 77).

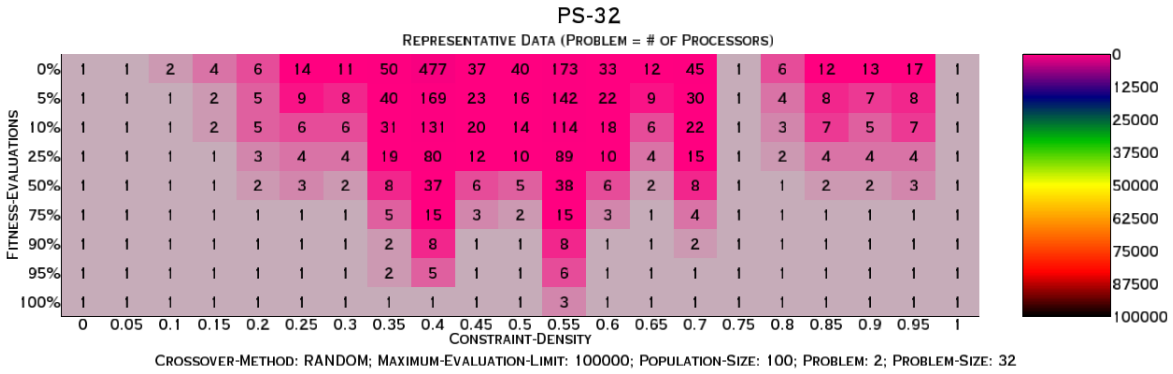

Figure 4.5.13: PS-32: 2 Processors

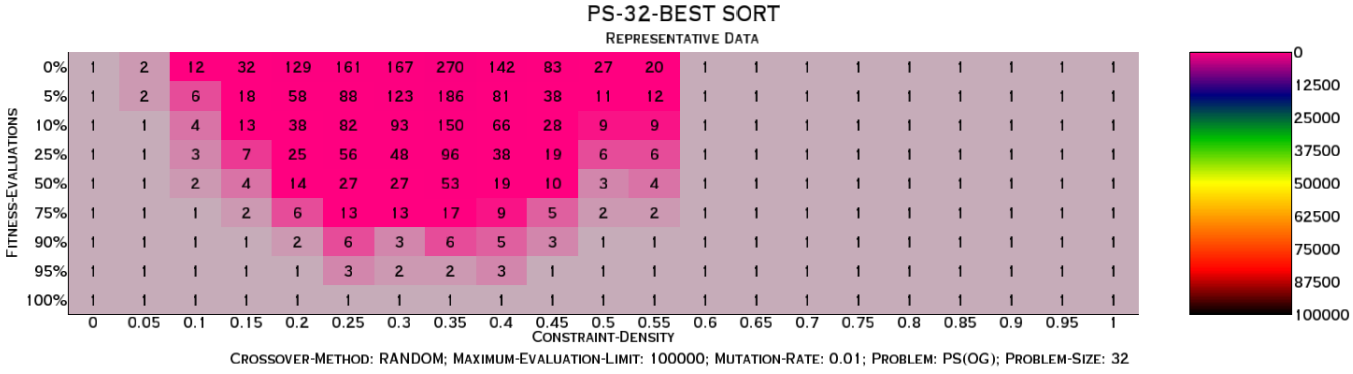

Figure 4.5.14: PS-32: 4 Processors

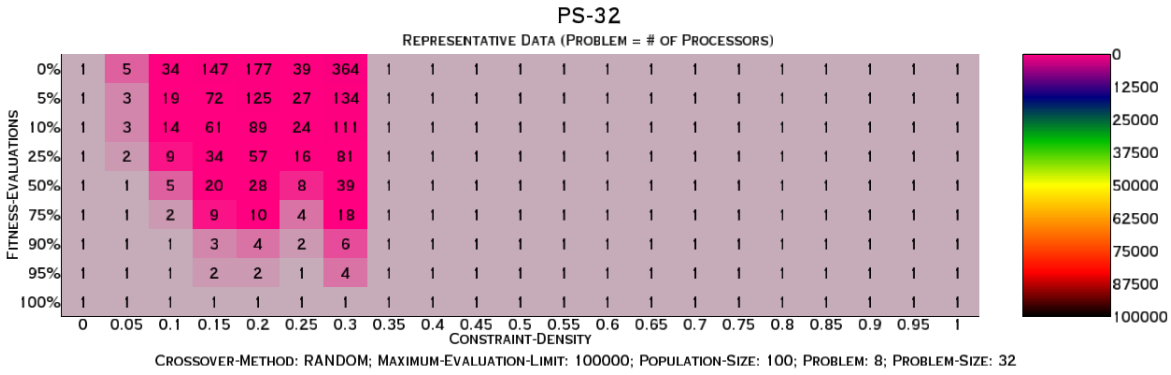

Figure 4.5.15: PS-32: 8 Processors

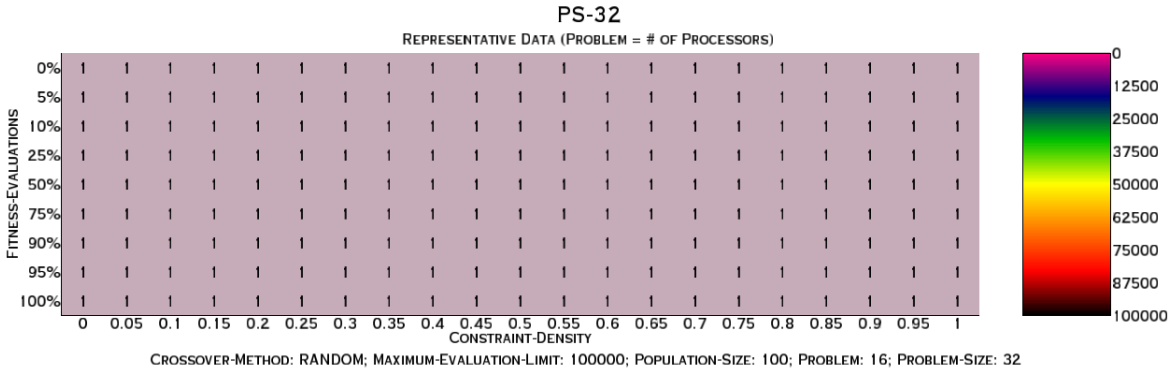

Figure 4.5.16: PS-32: 16 Processors

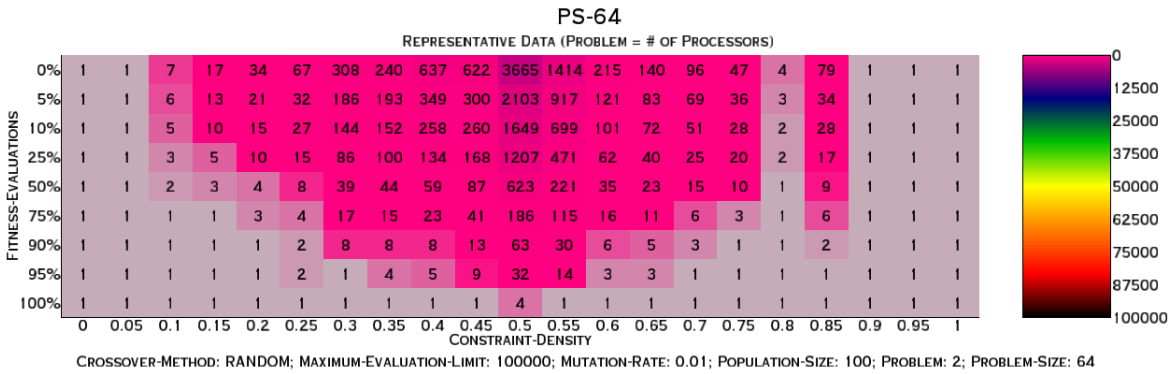

Figure 4.5.17: PS-64: 2 Processors

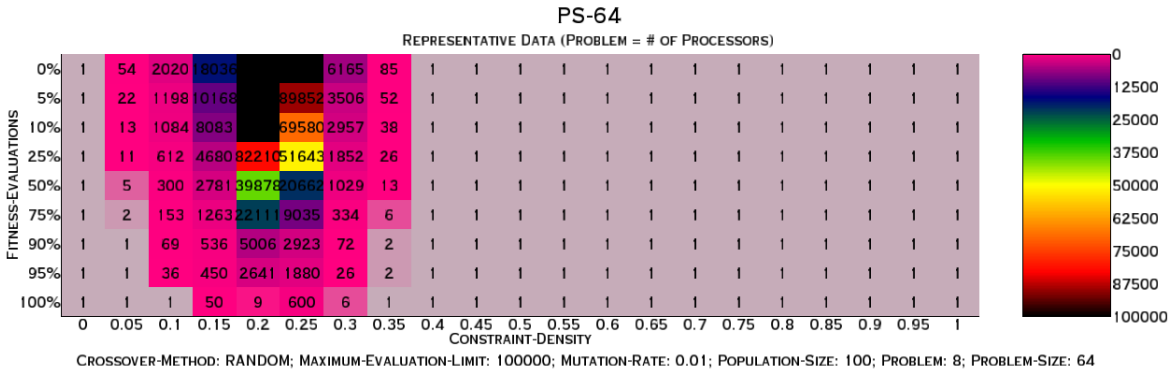

Figure 4.5.18: PS-64: 8 Processors

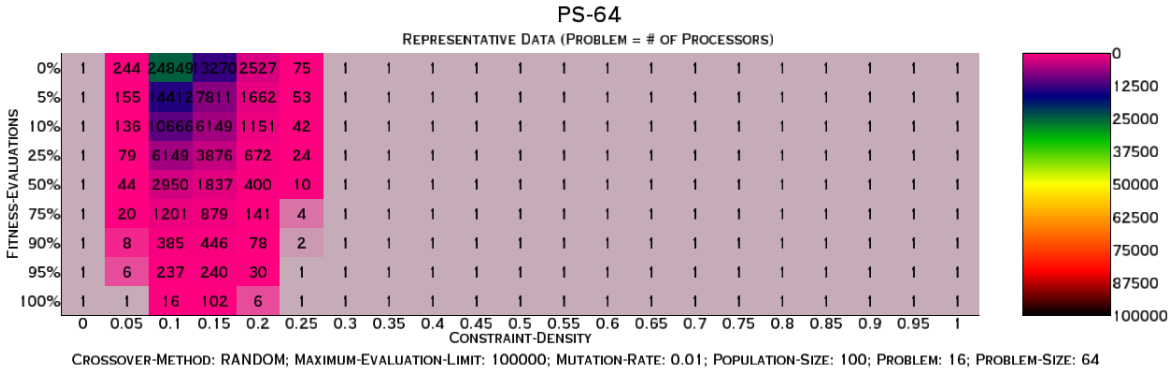

Figure 4.5.19: PS-64: 16 Processors

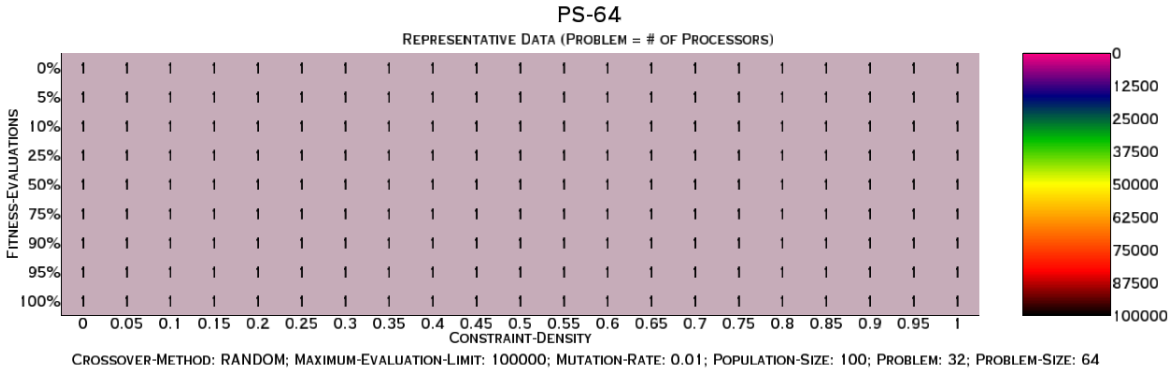

Figure 4.5.20: PS-64: 32 Processors

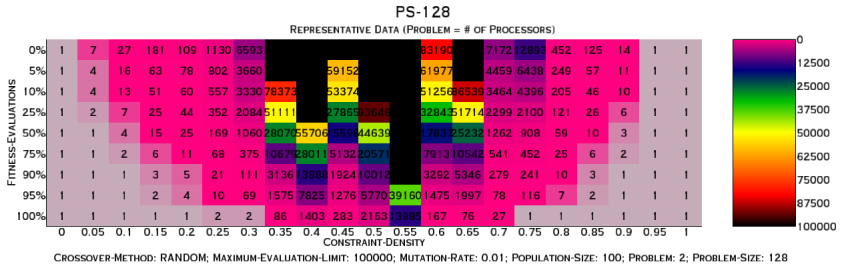

Figure 4.5.21: PS-128: 2 Processors

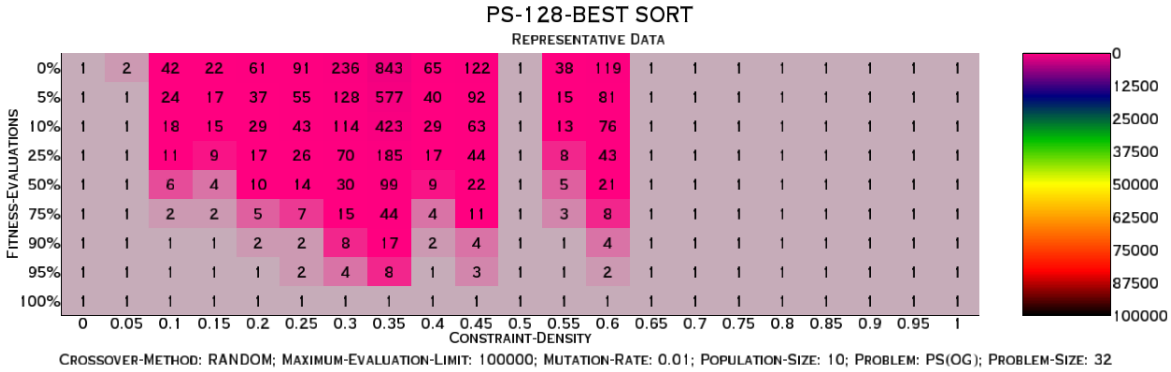

Figure 4.5.22: PS-128: 4 Processors

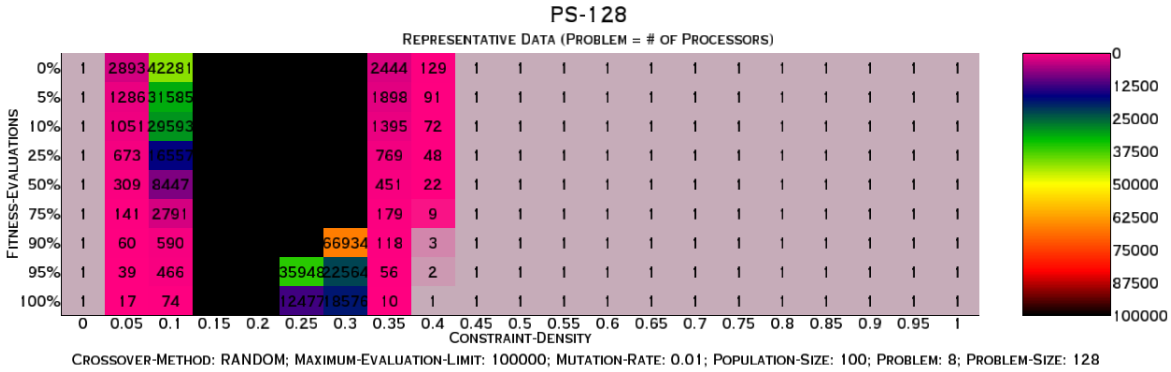

Figure 4.5.23: PS-128: 8 Processors

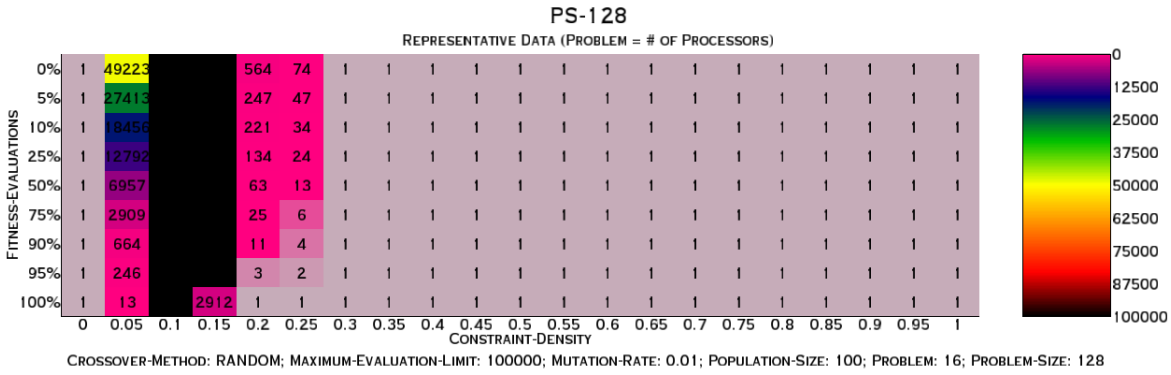

Figure 4.5.24: PS-128: 16 Processors

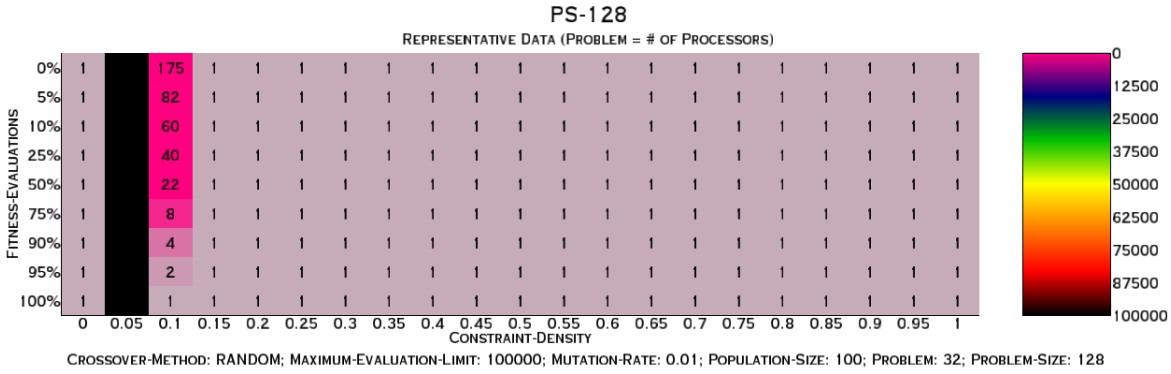

Figure 4.5.25: PS-128: 32 Processors

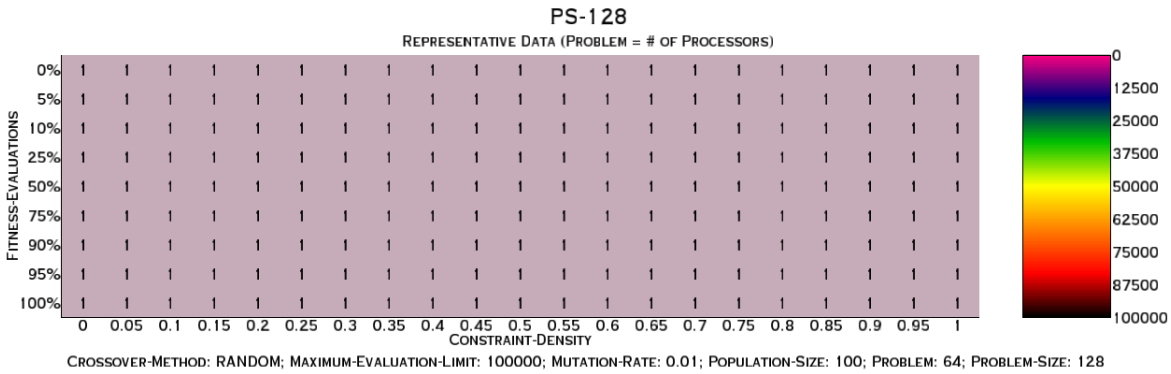

Figure 4.5.26: PS-128: 64 Processors

4.5.2 Crossover Techniques

4.5.2.1 Primary Results

From Subsection 4.5.1.2 on page 57 it is already evident that CX and PMX perform well. Additionally, by looking pairwise at Figures4.5.2.1 through 4.5.2.1, which depict performance at difficult constraint densities (0.15 through 0.3), their success is further evident, as well as SX-UNIFORM's.

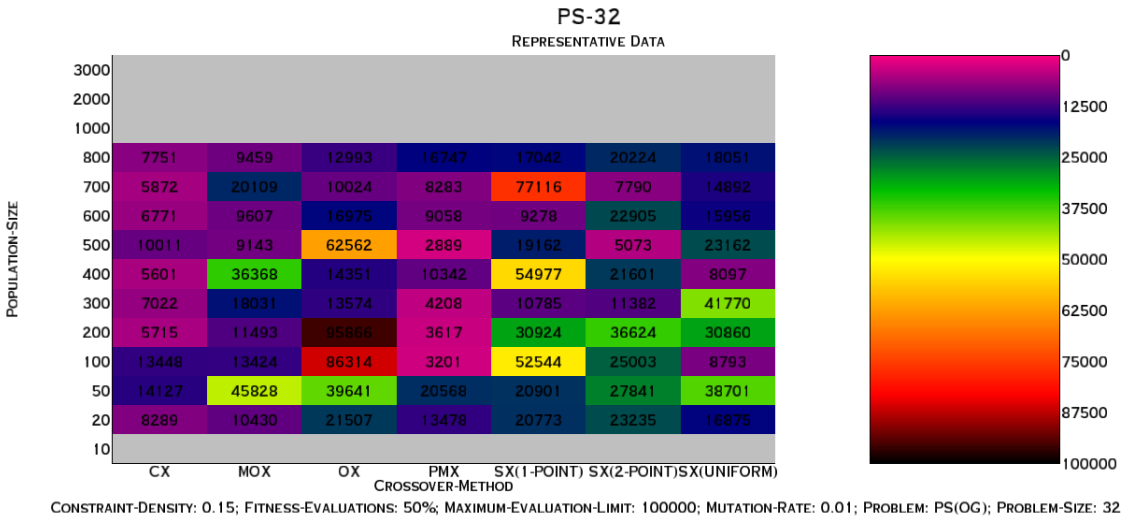

Figure 4.5.27: PS-32: 50th percentile data; constraint density 0.15

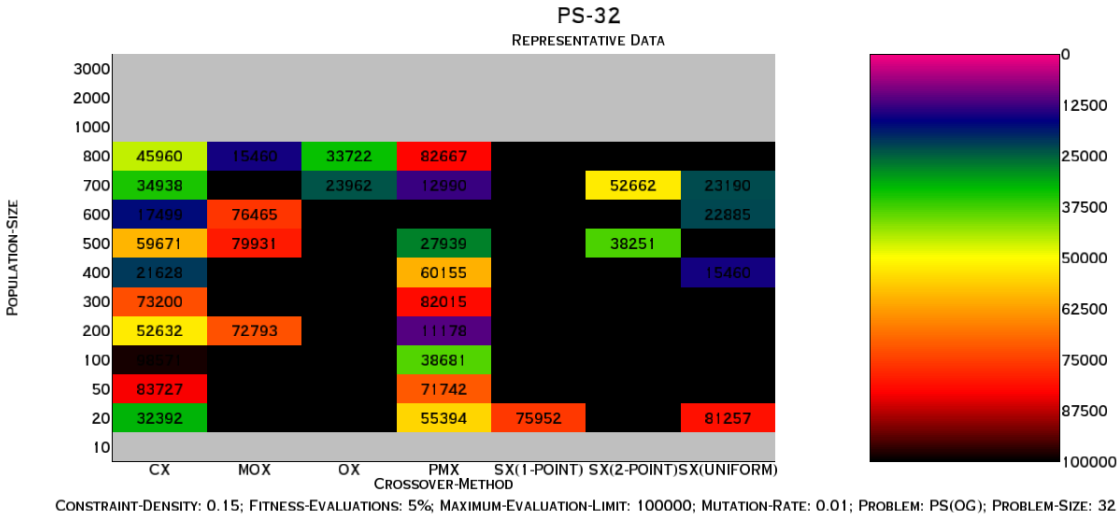

Figure 4.5.28: PS-32: 5th percentile data; constraint density 0.15

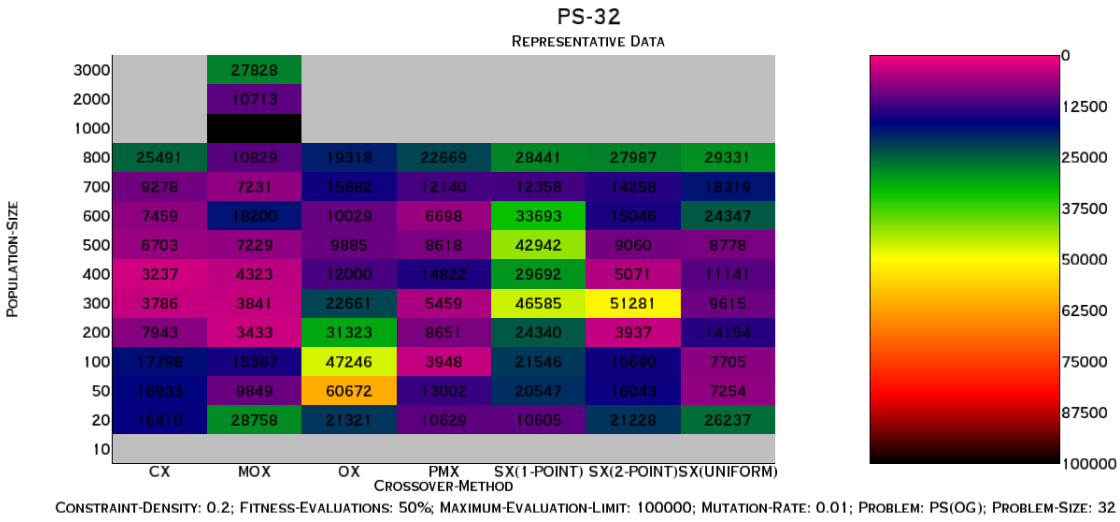

Figure 4.5.29: PS-32: 50th percentile data; constraint density 0.2

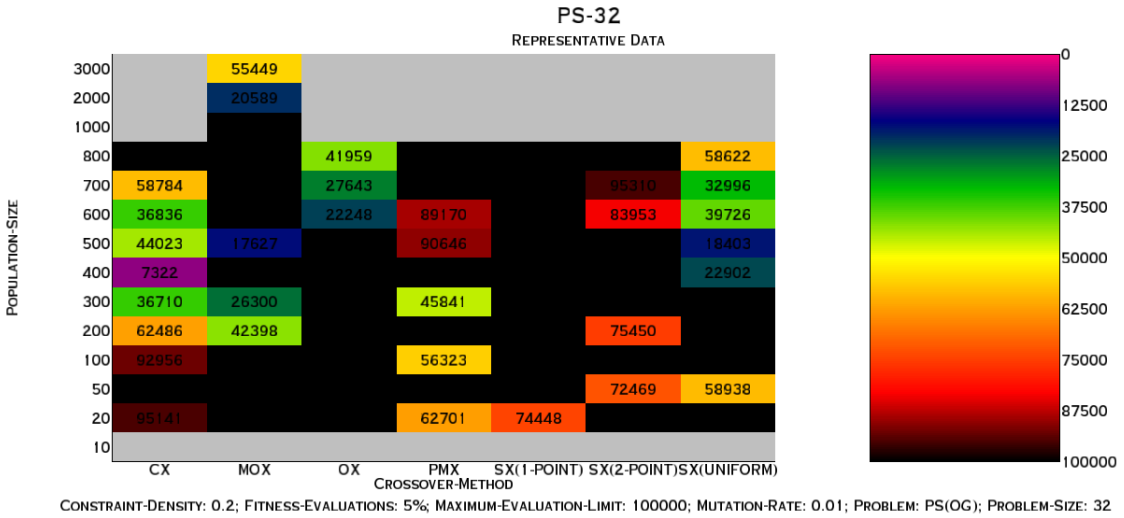

Figure 4.5.30: PS-32: 5th percentile data; constraint density 0.2

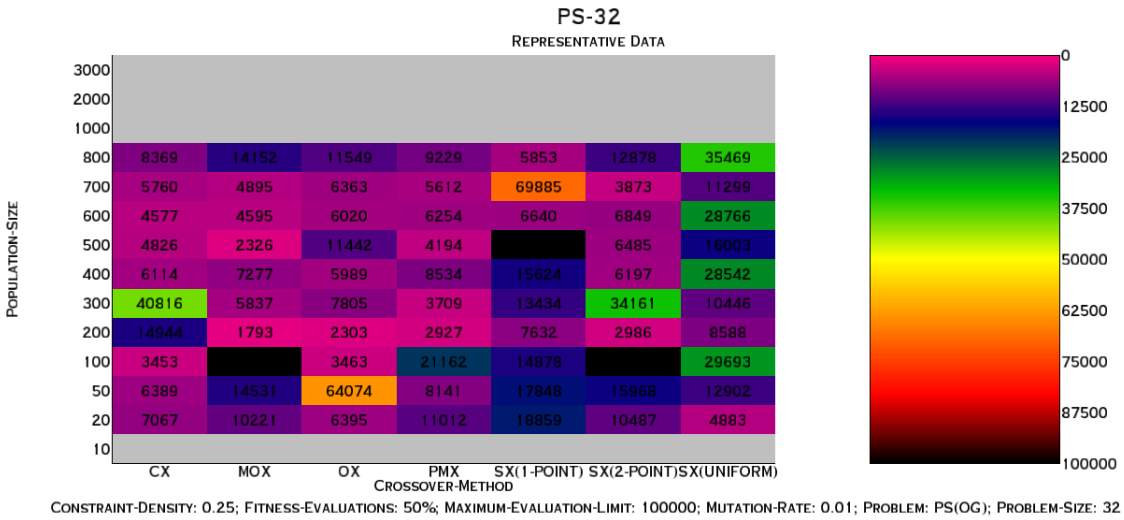

Figure 4.5.31: PS-32: 50th percentile data; constraint density 0.25

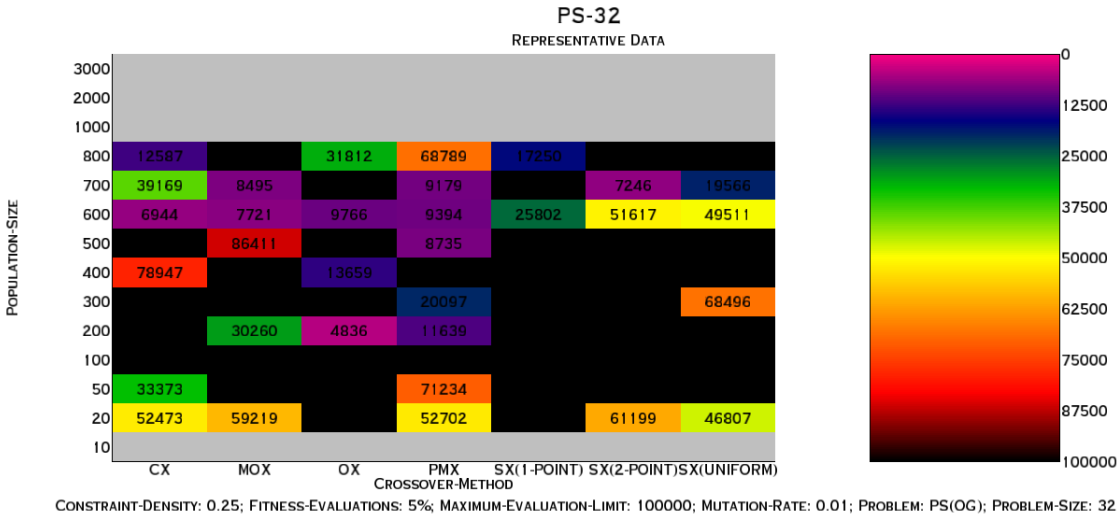

Figure 4.5.32: PS-32: 5th percentile data; constraint density 0.25

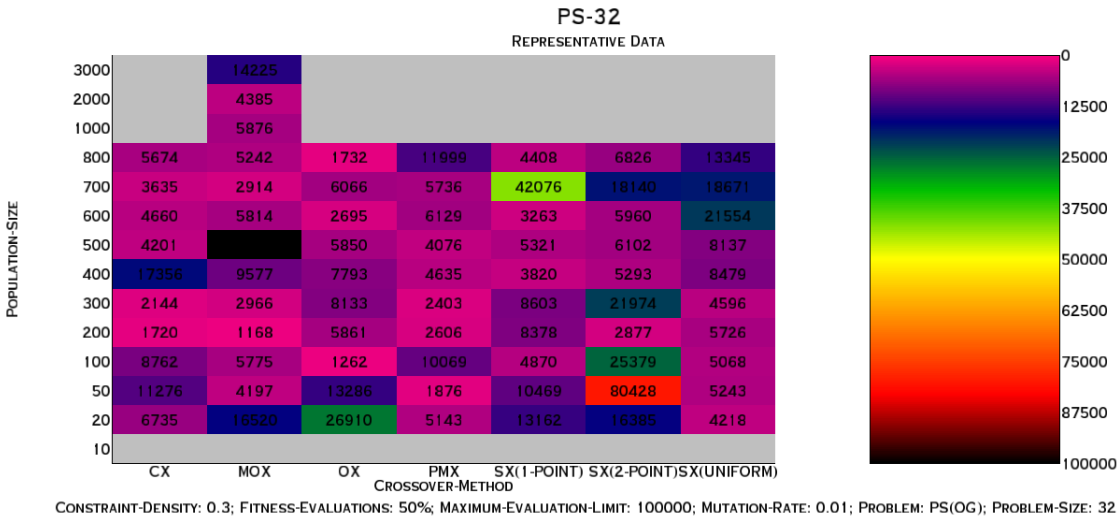

Figure 4.5.33: PS-32: 50th percentile data; constraint density 0.3

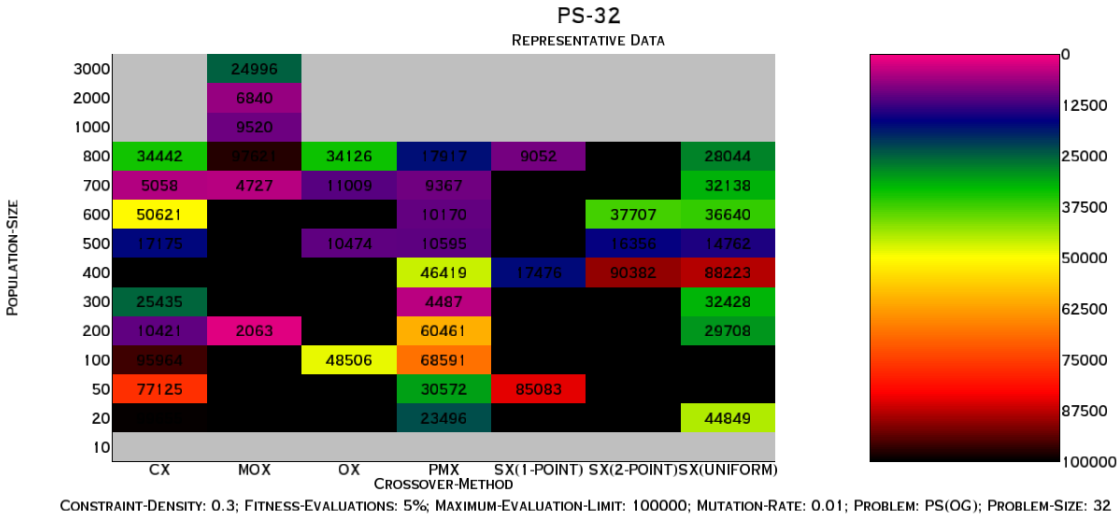

Figure 4.5.34: PS-32: 5th percentile data; constraint density 0.3

4.5.2.2 Effect of Population Size on GA Performance

The previous section already suggests that low to midrange population sizes are often successful. Table 4.5.2.2 on the next page, which highlights the best 20 parameter configurations further exemplifies this result. Of additional note may be that CX does not have great prominence in these charts, suggesting further that its success comes from solid average results without relying on outlying success to boost its numbers.

Constraint Density: 0.1		Constraint Density: 0.15		Constraint Density: 0.2	
SX(UNIFORM)	50	OX	700	MOX	20
MOX	600	PMX	500	MOX	600
PMX	300	CX	200	PMX	100
PMX	100	CX	300	PMX	100
SX(2-POINT)	100	PMX	50	MOX	800
MOX	100	PMX	200	CX	50
SX(2-POINT)	100	CX	200	SX(1-POINT)	50
SX(1-POINT)	500	SX(UNIFORM)	100	PMX	20
MOX	400	PMX	500	SX(2-POINT)	100
OX	20	PMX	500	PMX	600
MOX	300	PMX	300	MOX	200
SX(1-POINT)	300	MOX	300	PMX	200
SX(UNIFORM)	200	SX(UNIFORM)	400	PMX	700
CX	100	SX(1-POINT)	600	MOX	300
SX(1-POINT)	200	CX	400	CX	300
PMX	800	MOX	100	PMX	200
SX(1-POINT)	200	OX	200	SX(2-POINT)	400
PMX	400	PMX	100	MOX	200
MOX	700	MOX	400	MIX	200
CX	50	SX(UNIFORM)	100	SX(2-POINT)	100
Constraint Density: 0.25		Constraint Density: 0.3		Constraint Density: 0.35	
MOX	50	MOX	100	OX	600
CX	20	SX(1-POINT)	50	MOX	20
SX(2-POINT)	20	PMX	50	MOX	100
PMX	20	OX	800	CX	100
PMX	50	PMX	50	PMX	50
MOX	50	PMX	50	PMX	50
OX	200	MOX	300	PMX	50
OX	20	SX(UNIFORM)	20	CX	100
SX(2-POINT)	400	SX(UNIFORM)	20	SX(2-POINT)	20
MOX	500	SX(UNIFORM)	50	CX	20
PMX	400	SX(UNIFORM)	200	PMX	20
SX(UNIFORM)	800	PMX	100	PMX	700
CX	200	MOX	200	OX	50
MOX	500	CX	200	SX(2-POINT)	50
PMX	200	OX	200	MOX	400
MOX	200	CX	100	PMX	500
CX	400	MOX	50	MOX	200
SX(2-POINT)	200	OX	100	MOX	200
CX	300	SX(UNIFORM)	50	MOX	300
SX(1-POINT)	20	PMX	20	SX(1-POINT)	50

Table 4.5.3: Best 20 crossover method and population size configurations (sorted by 50th percentile values, across all GA runs) for selected constraint densities

4.5.2.3 Effect of Sorting Algorithm on GA Performance

The effect of presorting PS input is profound, and the benefit of a full sort exceeds the overhead required to implement such sort, at least within the methods explored in this research. Compare the results from Figures 4.5.2.3 to 4.5.2.3 against the primary results of Subsection 4.5.1.1 on page 54. In all cases, the fully sorted data severely outperforms the roughly sorted primary data.

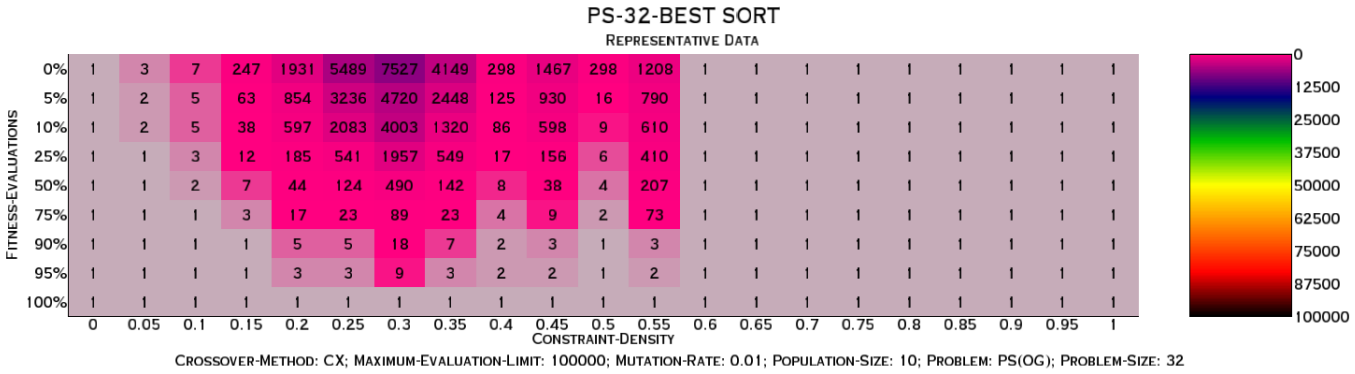

Figure 4.5.35: PS-32: Full sort, CX at population size 10

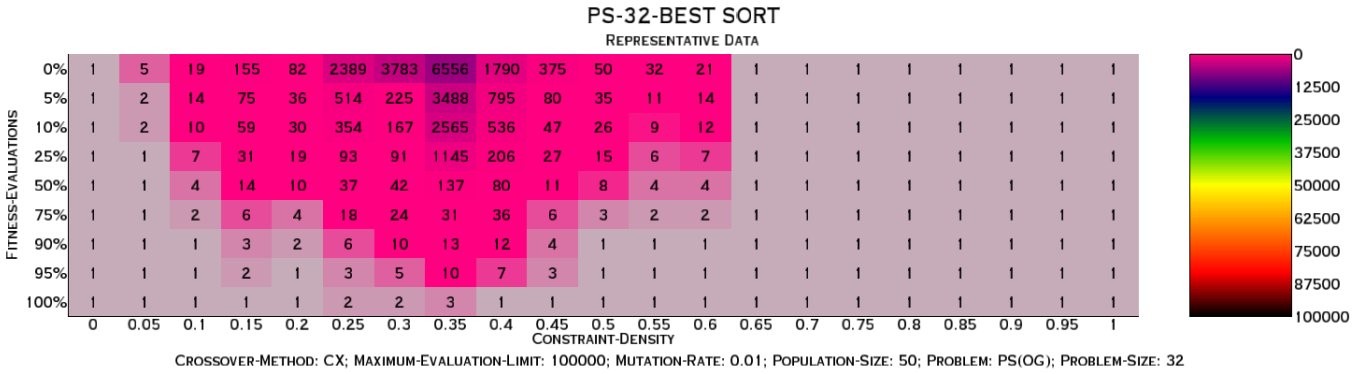

Figure 4.5.36: PS-32: Full sort, CX at population size 50

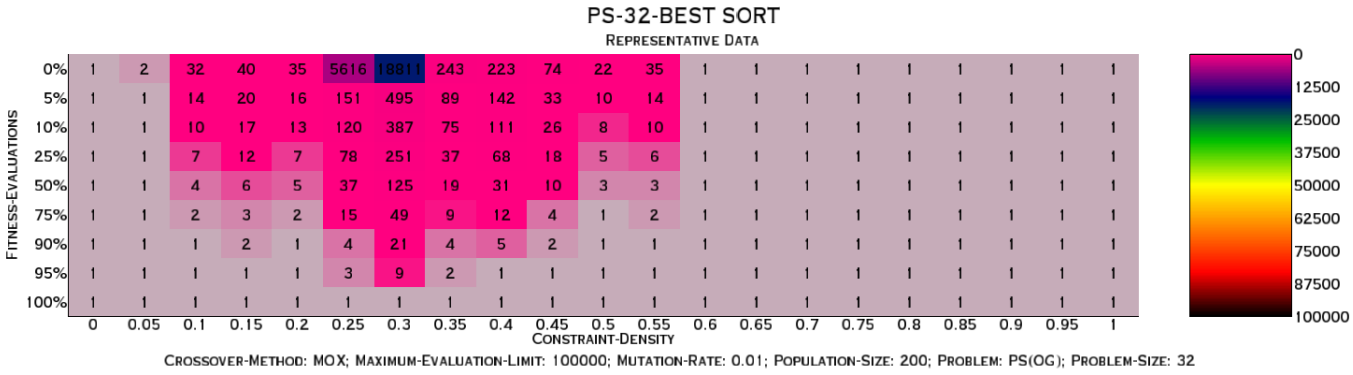

Figure 4.5.37: PS-32: Full sort, MOX at population size 200

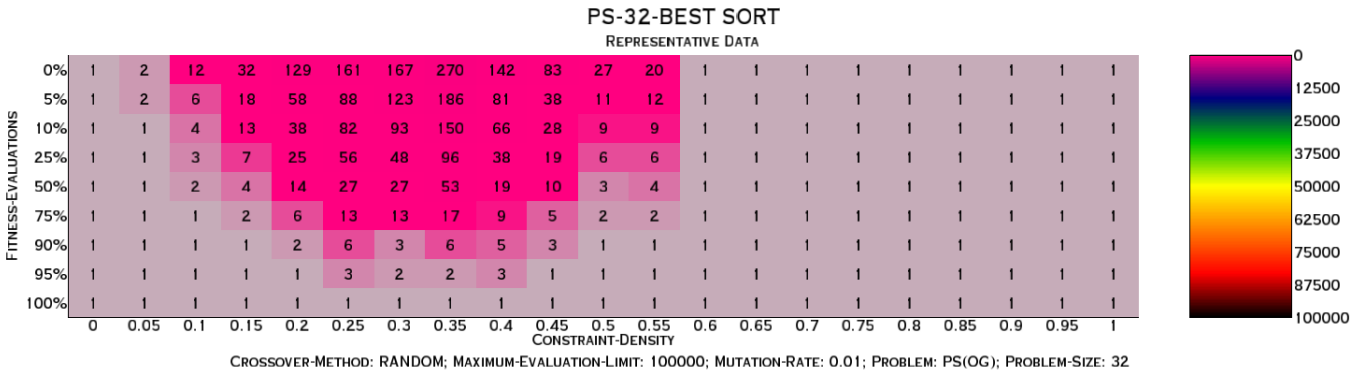

Figure 4.5.38: PS-32: Full sort, RANDOM search

It is also of immediate note that the RANDOM search outperforms the genetic algorithm when operating on fully sorted data, at least for the ranges of parameters tested. This is not necessarily a result of the GA being an inappropriate methodology as much as it shows how immensely powerful the sort is. Especially given the extent to which this sort would inevitably reorder the input, it remains a potential area of future research (see Subsection 5.2 on page 77) to determine if may be exploitable characteristics of fully sorted data that would still be amenable to a GA approach.

Unsorted data, not surprisingly, performs much worse than any method of sort. Consider Figure 4.5.2.3, which summarizes the availability of some scattered unsorted data. Even with such incomplete data, it is clear that the unsorted data is far inferior

to even though roughly sorted input.

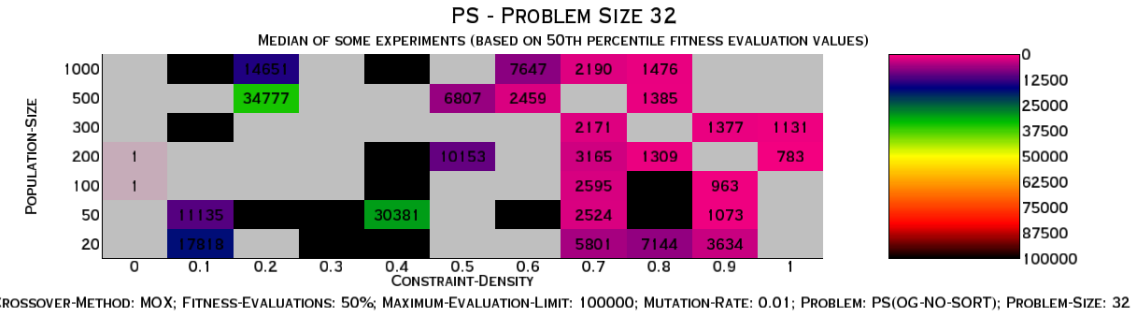

Figure 4.5.39: PS-32: MOX-200; Unsorted input at 50th percentile of quality

Data that utilized the partially random sort described in Subsection 4.4.1.3 on page 54 did have a considerable effect as can be seen in Figure 4.5.2.3, and did also notably outperform the roughly sorted data seen previously in Figure 4.5.1.3 on page 58.

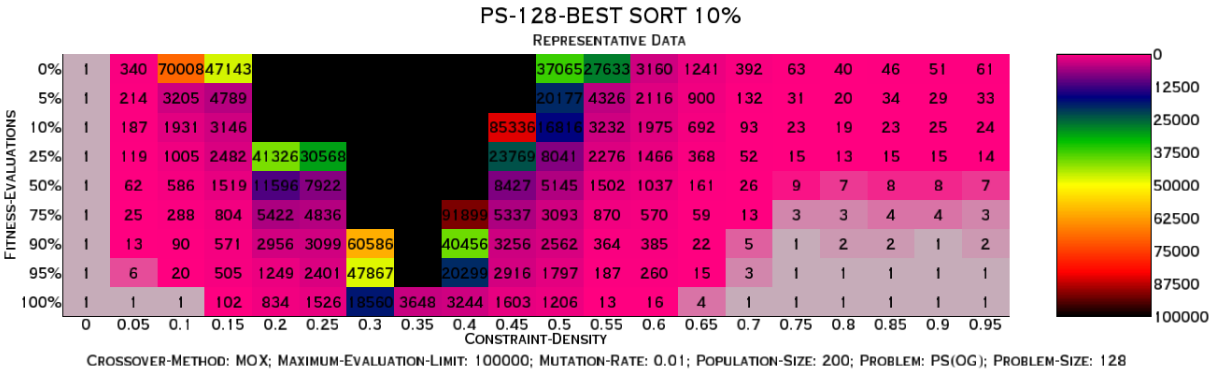

Figure 4.5.40: PS-128: Partially random sort, MOX at population size 200

To compensate for the fact that a full sort should be expected to, even without much algorithmic study, require more time to complete than other forms of sort performed in this research, a true time analysis that measured sorting performance in terms of fitness evaluations performed per second. For the PS-64 problem at a constraint density of 0.3, full sort performed at roughly an average of 3155 fitness evaluations per second, rough sort performed at roughly 5995 fitness evaluations per second, partially random at 17,775 fitness evaluations per second, and unsorted at 27,070. While the

full sort performed, in real time, several times slower than other methods, it often performed hundreds, even thousands of times better. Also considering such a time comparison, it is also evident that the partially random sort outperforms the rough sort as well.

Chapter 5

Conclusion

5.1 Implications of Research

In addition to the results highlighted in each section, there are several cumulative conclusions to be made from this research.

- Determining winners and losers, especially in terms of crossover method and population size, cannot always be clear-cut.
- Investigation of constraint density definitively showed exact values or narrow ranges at which problems are most often difficult. However, difficult problems can still appear across wide ranges of parameters. Likewise there is not always an obvious set of baseline parameters that will result in difficult problems.
- Alterations of population size, crossover technique, and problem size do not seem to make vast differences in problem difficulty. Further research into problem difficulty may not need to examine such things so closely except when the goal is truly to exploit all possible GA parameters to maximize performance.

5.2 Future Work

5.2.1 Additional In-Depth Research

There are several avenues upon which futher related research could be pursued:

- Testing other GA parameters or options, such as selection method, population representation, or more work with mutation rate
- Implementing a crossover strategy or customized sort that directly attempts to reduce the difficulty of NQWP problems or attacks difficult problems in a pointed manner
- Analysis of particular qualities of difficult PS problems, and to better understand how and why particular processor counts so greatly affects where problems are difficult
- Determining if there exist any ways to improve GA performance of fully sorted PS data
- How results compare to non-OG or non-permutation based approaches to the same problems
- Consider a more graph-theoretical approach for NQWP and then determine if any graph theory can be exploited to better solve problems
- If greater computational resources (and/or time) are available, run experiments described in this thesis more thoroughly, and/or examine constraint densities with more granularity than a delta of 0.05

5.2.2 Additional Problems

There are many other problems along the same vein as NQWP and PS that could be examined in a similar fashion, as well as variations on NQWP and PS. Some of these possibilities are listed as follows.

5.2.2.1 Graph Coloring

A k-partite graph is a graph that can be broken into k subgraphs, where vertices within each subgraph are unconnected by edges. For example, a tripartite (3-partite) graph can be broken into three such subgraphs. Further, a complete k-partite graph is one where each vertex in the aforementioned subgraphs is connected by edges to all vertices outside of its respective subgraph. Again using the complete tripartite graph as an example, the notation $K_{r,s,t}$ is often used to represent its topography, where r, s, and t specify the number of vertices in each subgraph [5]. Because tripartite graphs can be broken down in this way, they may be colored using as few as three colors (such that no two vertices of the same color are connected by any edge), a point that can be extended for any positive value of k. For this problem, k-coloring will be attempted on graphs that are subsets of complete k-partite graphs of the form $K_{\underbrace{n, n, ..., n}_{k}}$.

5.2.2.2 Classroom Seating

In a class of n students to be assigned seats in one row of n desks (or a matrix of $m \times n$ desks), there would be many things a teacher may want to take into account, including and especially students what should not sit near each other as their proximity would be likely to disrupt the class. The CS problem involves taking a list of such constraints

as pairs of students and finding a seating arrangement where such pairings do not occur within a conflict radius of k desks of each other.

5.2.2.3 Variations on *N*-Queens With Poison

Apply the same principal to packing knights onto a poisoned board, or devising other pieces which may be even more difficult to place than queens, etc.

5.2.2.4 Variations on Processor Scheduling

Extend the definition of the problem to include tasks that take longer than one unit of time to complete.

Appendix A

Derivation of Average OX/PMX Crossover Section Size

In selecting p_1 and p_2 uniformly over n, there are n choices for both p_1 and p_2 and to assume without loss of generality that $p_1 \leq p_2$, it suffices for work with the crossover techniques to swap p_1 and p_2 if $p_1 > p_2$. The following chart shows the crossover section sizes for each value of p_1 and p_2.

		p_1					
		0	1	2	$\cdots$	$n-2$	$n-1$
p_2	0	1	2	3	$\cdots$	$n-1$	n
	1	2	1	2	$\cdots$	$n-2$	$n-1$
	2	3	2	1	$\cdots$	$n-3$	$n-2$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
	$n-2$	$n-1$	$n-2$	$n-3$	$\cdots$	1	2
	$n-1$	n	$n-1$	$n-2$	$\cdots$	2	1

Table A.0.1: Length of crossover section sizes in OX/PMX for values of p_1 and p_2

The average crossover section size will be computed by adding all of the individual crossover section sizes and then dividing that value by the total number of crossover sections. It is immediately clear that there are n^2 (not necessarily unique) crossover sections. There are n sections of size 1, $2(n-1)$ sections of size 2, $2(n-2)$ sections of size 3, etc., 2(2) sections of size $n-1$, and finally 2 sections of size n. This equates

to $n \cdot 1 + 2(n-1) \cdot 2 + 2(n-2) \cdot 3 + ... + 2(2) \cdot (n-1) + 2(1) \cdot n$, or, to uniformly have a coefficient of 2 in all sequential terms,

$$-n + 2(n) \cdot 1 + 2(n-1) \cdot 2 + 2(n-2) \cdot 3 + ... + 2(2) \cdot (n-1) + 2(1) \cdot n =$$

$$2[1(n) + 2(n-1) + 3(n-2) + ... + (n-1)(2) + n(1)] - n =$$

$$2\big[n \cdot (1 + 2 + 3 + ... + n - 1 + n) - [0 + 2 + 6 + ... + (n-1)(n-2) + n(n-1)]\big] - n =$$

$$2[n \cdot \tfrac{n(n+1)}{2} - 2(0 + 1 + 3 + ... + \tfrac{(n-1)(n-2)}{2} + \tfrac{n(n-1)}{2})] - n =$$

$$2\left[\tfrac{n^2(n+1)}{2} - 2\left(\tbinom{1}{2} + \tbinom{2}{2} + \tbinom{3}{2} + ... + \tbinom{n-1}{2} + \tbinom{n}{2}\right)\right] - n =$$

$$2\big[\tfrac{n^2(n+1)}{2} - 2\tbinom{n+1}{3}\big] - n =$$

$$n^2(n+1) - \tfrac{4(n-1)(n)(n+1)}{6} - n =$$

$$n^2(n+1) - \tfrac{2(n-1)(n)(n+1)}{3} - n =$$

$$n(n+1)[n - \tfrac{2}{3}(n-1)] - n =$$

$$n(n+1)(\tfrac{1}{3}n + \tfrac{2}{3}) - n =$$

$$\tfrac{1}{3}n(n+1)(n+2) - n$$

Finally, the average avg can now be computed as $\frac{\frac{1}{3}n(n+1)(n+2)-n}{n^2} = \frac{\frac{1}{3}(n+1)(n+2)-1}{n} = \frac{\frac{1}{3}(n^2+3n+2)-1}{n} = \frac{1}{3}n + 1 + \frac{2}{3n} - \frac{1}{n} = \frac{n}{3} + 1 - \frac{1}{3n}$. Since $0 < n < \infty$ and $\lim_{n\to\infty} \frac{1}{3n} = 0$, $\frac{n}{3} + \frac{2}{3} \leq avg < \frac{n}{3} + 1$. In other words, $avg \approx \frac{n}{3}$.

Bibliography

[1] Peter G. Anderson, Daniel Ashlock: *Advances in Ordered Greed* in Intelligent Engineering Systems Through Artificial Neural Networks, Vol 14:223-228 (2004)

[2] S. N. Sivanandam, S. N. Deepa: *Introduction to Genetic Algorithms*; Springer. Berlin, Germany (2007)

[3] Douglass Huang: *Assignment 5: Minimum Precedence Constrained Scheduling with Ordered Greed* (2011)

[4] Wilhelm Erben: *Genetic Algorithms*, Lecture held at School of CS & IT, The University of Nottingham (March 2006)

[5] Eric W. Weisstein: *Complete k-partite Graph*; From MathWorld–A Wolfram Web Resource (http://mathworld.wolfram.com/Completek-PartiteGraph.html)

[6] Dennis Coon, John O. Mitterer: *Introduction to Psychology: Gateways to Mind and Behavior*; Wadsworth. Belmont, California (2010)

[7] Richard Allen King, Jerome I. Rotter, Arno G. Motulsky: *The Genetic Basis of Common Diseases*; Oxford University Press, Inc. Oxford (2002)

[8] Frederic P Miller, Agnes F Vandome, John McBrewster: *Chromosomal Crossover*; VDM Publishing House Ltd. (2009)

[9] T. Strachan, Andrew P. Read: *Human Molecular Genetics 3*; Garland Science. New York (2004)

[10] Charles Darwin: *On the Origin of Species*; John Murray. London (1859)

[11] Erik D. Goodman: *Introduction to Genetic Algorithms*, Presentation given at 2009 World Summit on Genetic and Evolutionary Computation, Shanghai, China.