Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

2006

Metaheuristics and combinatorial optimization problems

Gerald Skidmore

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Skidmore, Gerald, "Metaheuristics and combinatorial optimization problems" (2006). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Metaheuristics and Combinatorial Optimization Problems

by

Gerald Skidmore Scriptor

B.S. University of Buffalo, 2000

A thesis submitted to the

Faculty of the Graduate School of the

Rochester Institute of Technology in partial

fulfillment of the requirements for the degree of

Masters of Science in Computer Science

Department of Computer Science

2006

Rochester Institute of Technology

This thesis entitled:
Metaheuristics and Combinatorial Optimization Problems
written by Gerald Skidmore Scriptor
has been approved for the Department of Computer Science

_____

Peter Anderson

_____

Phil White

_____

Stanislaw P Radziszowski

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Scriptor, Gerald Skidmore (M.S., C.S.)

Metaheuristics and Combinatorial Optimization Problems

Thesis directed by Professor Peter Anderson

This thesis will use the traveling salesman problem (TSP) as a tool to help present and investigate several new techniques that improve the overall performance of genetic algorithms (GA). Improvements include a new parent selection algorithm, harem select, that outperforms all other parent selection algorithms tested, some by up to 600%. Other techniques investigated include population seeding, random restart, heuristic crossovers, and hybrid genetic algorithms, all of which posted improvements in the range of 1% up to 1100%.

Also studied will be a new algorithm, GRASP, that is just starting to enjoy a lot of interest in the research community and will also been applied to the traveling salesman problem (TSP). Given very little time to run, relative to other popular metaheuristic algorithms, GRASP was able to come within 5% of optimal on several of the TSPLIB maps used for testing. Both the GA and the GRASP algorithms will be compared with commonly used metaheuristic algorithms such as simulated annealing (SA) and reactive tabu search (RTS) as well as a simple neighborhood search - greedy search.

Contents

Chapter

Tables

Table

Figures

Figure

Algorithms and Theory

0.1 Introduction

Optimization problems involving a large number of finite solutions often arise in academia, government, and industry. In these problems there is a finite solution set X and a real-valued function $f : X \rightarrow \mathbf{R}$ where we seek a solution $x* \in X$ with $f(x*) \leq f(x)$, $\forall x \in X$. Common examples include crew scheduling, vehicle routing, and VLSI routing. To find the globally optimal solution in a combinatorial optimization (CO) problem it is theoretically possible to enumerate all possible solutions and evaluate each. This approach is generally undesirable as well as intractable due to the exponential growth of most solution spaces.

A simple search of any technical database will quickly demonstrate that a lot of research has been devoted over the past five or so decades to find and develop optimal or near-optimal algorithms that can quickly and efficiently find a solution without evaluating every possible solution. Through this research have arisen the fields of combinatorial optimization and meta-heuristics along with the abilities to solve ever larger problem instances. Metaheuristics evolved because most modern problems are computationally intractable, needing heuristic guidance to find good solutions, but not necessarily the most optimal. Some of the most promising techniques include genetic algorithms (GA), Greedy Randomized Adaptive Search Procedure (GRASP), simulated annealing (SA), and reactive tabu search (RTS).

In this thesis, I present some new extensions to genetic algorithms, explore the GRASP algorithm, and compare these to some of the more commonly employed techniques of simulated annealing, reactive tabu search, and greedy algorithms. The components of a basic GRASP heuristic are discussed and enhancements proposed to the basic heuristic are discussed. Following this format, genetic algorithms are discussed in their most basic form as well as several improvements. The paper concludes with an overview of the previously said algorithms and the performance reviews of all as applied to the traveling salesman problem (TSP).

Enhancements to the genetic algorithm will include new parent selection techniques that

improve the overall performance by as much as 500% when compared to the most commonly used techniques such as tournament select and roulette select. In addition to the new parent selection algorithm, I will present 2 new concepts I refer to as adaptive mutation and adaptive restart. All of these techniques will be compared for performance against each other as well as some of the other commonly employed metaheuristic algorithms such as simulated annealing (SA) and the reactive tabu search (RTS).

GRASP, or greedy randomized adaptive search procedure, is a newer but lesser known algorithm that will also be presented and tested against the TSP. GRASP's results often came within 5% of the known optimal solution; even more impressive is that GRASP usually required only a fraction of the running time of the others.

0.2 Traveling Salesman Problem

Given a set of cities and the distances between them, the goal of the traveling salesman problem (TSP) is to find a complete, minimal cost tour that visits each city once and returns to the starting city. The TSP is a well-known NP-hard problem with many real-world applications, such as VLSI, job shop scheduling, and delivery route planning. The TSP has often served as a jumping block for new problem-solving techniques and algorithms. Many well-known combinatorial algorithms were first developed for the TSP, including the Lin-Kernighan local search algorithm [22]. In this thesis, I restrict the problem scope to the symmetric, euclidean 2-D TSP.

An important problem in several areas, TSP has long been known to be in the class of NP-Complete problems, often the standard example given because of its practical and theoretical value. According to Applegate, et. al., TSP's prominence in the literature "is to a large extent due to its success as an engine-of-discovery for techniques that have applications far beyond the narrow confines of the TSP itself." [7] It is for this very reason that I employ the TSP - it will help in gauging the performance of the algorithms against a well known and studied body of work.

0.3 TSP Background

An extremely deceptive problem, the traveling salesman problem (TSP) states given N cities, $c_1, \ldots, c_n$, and the distances between them, $d(c_i, c_j)$, find the shortest path that visits every city and returns to the starting city. Even knowing the cities that need to be visited and the distances between them, finding the path that minimizes the distance traveled is not as easy as it sounds.

So what can be said about solution methods for the TSP? It is easy to develop brute-force methods for this problem that will have a running time of $(n-1)!$. Adding a little bit of memory to these "dumb" algorithms can improve these bounds from $(n-1)!$ to $(n-1)!/2$ since the tour 3,2,1 is the same as the tour 1,2,3. But as Table 1 demonstrates, neither of these bounds are desirable! Michael Held and Richard Karp [16] presented an algorithm in 1962 that guarantees a bound of $n^2 2^n$. It is easy to see that for larger values of n, the Held-Karp bound is significantly better than $(n-1)!$. Unfortunately, in the over 40 years since, no one has been able to find a better lower bound!

In fact, most of the best algorithms put forth to solve this problem were done quite a while ago! The Lin-Kernighan (LK) algorithm was developed thirty years ago, but it is still considered to be one of the best heuristics for the Euclidean TSP! The simplex method for finding lower bounds was presented in the 1950's. TABU and Simulated Annealing are both over 20 years old as is true for genetic algorithms. What have been changing though are the improvements made to all of the aforementioned algorithms, as this thesis is continuing.

The TSP problem, because of its extreme difficulty, is central to the study of combinatorial optimization (CO). CO problems are typified by a search for an object in a finite set or countably infinite set of objects. This object can be a number, a permutation, or something entirely different.

As more cities are added, the search space of the problem increases factorial, that is $N!$. A computer algorithm capable of searching the entire TSP solution space for fifty cities would

require an increase in power of 10^{40} just to add an additional ten cities. Clearly the "brute force" algorithm mentioned before ($(n-1)!$) becomes impossible for any problem instance with a large number of cities.

Table 1: This table demonstrates the problem of solving the TSP

Number of Cities	Size of Search Space
1	1
2	1
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
...	...
75	$2.5 \cdot 10^{109}$

To make the last entry in Table 1 a little more concrete, 75! = 24,809,140,811,395,398,091,-946,477,116,594,033,660,926,243,886,570,122,837,795,894,512,655,842,677,572,867,409,443,815,424,-000,000,000,000,000,000.

Since very few good solutions abound for solving this problem, the research community has developed a standard set of problem instances that are used by individual researchers to test against. This helps in making comparisons against wildly different algorithms a little more uniform. Currently, the repository, TSPLIB, can be located at [20]. The algorithms presented were tested against the Euclidean 2D maps available from TSPLIB in the following sizes: 52, 76, 225, 280, 299, 493, 657, 1291, 2103, 3795, 5915, and 13059.

0.3.1 Lower Bounds

Reading through much of the available TSP literature, one will inevitably come across researchers talking about lower bounds for the problem instances in the TSPLIB, or just problem instances in general. So how can we be certain a solution is optimal or within a certain percent

of optimality when even a 75 city problem has approximately $2.5 \cdot 10^{109}$ possible solutions?

The answer is a rather elegant solution put forth by G. Dantzig, R. Fulkerson, and S. Johnson [8] in the middle of the last century! They used linear programming to minimize a group of constraints to come up with a lower bound solution. The following comes from [6] where there is a much more in-depth coverage of the concepts I will lightly touch upon in the following examples and paragraphs.

Suppose for example we have a 6 city TSP map. We would draw circles of radius r_{1-6} centered around our 6 cities such that the circles do not overlap. Refer to Figure 1 for a single city and Figure 2 for how a small 6 city map would be drawn out.

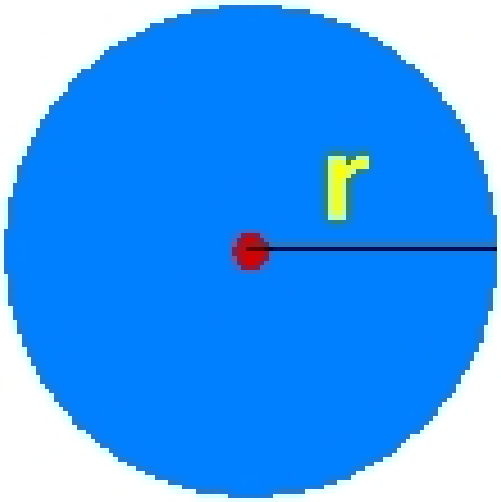


Figure 1: City With Radius.

At one point in the path, the salesman will have to travel to **and** leave the city centered within each of the circles as depicted in Figure 1. By doing so, they will have traveled a distance of $2r$ and every conceivable tour must have a minimum distance of $2r$!

Since we do not want the circles to overlap, they must be chosen in a way that maximizes their distance. At this point it will be easier to understand if we introduce some mathematical notation (previously introduced to describe the distance computations for the TSP): For a pair of cities i and j, dist(i,j) will be used to denote the distance from i to j. Choosing the radius

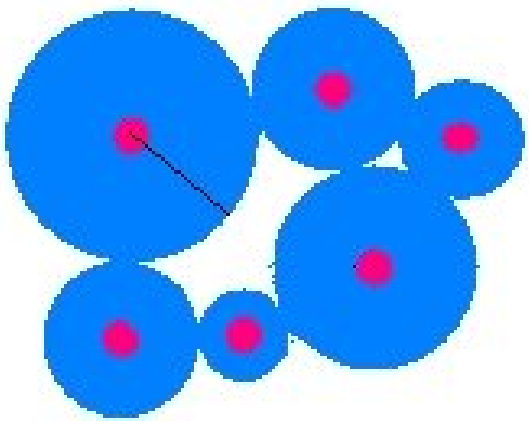


Figure 2: Six cities with radius zones that do no overlap.

such that $(r_i + r_j) \leq dist(i, j)$ will guarantee us that the disks will not overlap. Refer to Figure 2 for a simple example of this done with 6 cities.

Putting it all together, getting the best TSP bound for our cities can be written as follows:

$Maximize(2r_1 + 2r_2 + 2r_3 + 2r_4 + 2r_5 + 2r_6)$

Subject to the following 15 ($15 = \binom{n}{k} = \binom{6}{2}$) constraints:

$r_1 + r_2 \leq dist(1, 2)$

$r_1 + r_3 \leq dist(1, 3)$

$r_1 + r_4 \leq dist(1, 4)$

$r_1 + r_5 \leq dist(1, 5)$

$r_1 + r_6 \leq dist(1, 6)$

$r_2 + r_3 \leq dist(2, 3)$

$r_2 + r_4 \leq dist(2, 4)$

$r_2 + r_5 \leq dist(2, 5)$

$r_2 + r_6 \leq dist(2, 6)$

$r_3 + r_4 \leq dist(3, 4)$

$r_3 + r_5 \leq dist(3, 5)$

$r_3 + r_6 \leq dist(3, 6)$

$r_4 + r_5 \leq dist(4, 5)$

$r_4 + r_6 \leq dist(4, 6)$

$r_5 + r_6 \leq dist(5, 6)$

$r_1 \geq 0, r_2 \geq 0, r_3 \geq 0, r_4 \geq 0, r_5 \geq 0, r_6 \geq 0$

This is an example of a linear programming (LP) problem since we are maximizing a weighted sum, the radius of each of the disks, subject to the constraints that the disk's enveloping the cities don't overlap (all but the last constraint listed above) and that no disk should have a negative radius (last constraint).

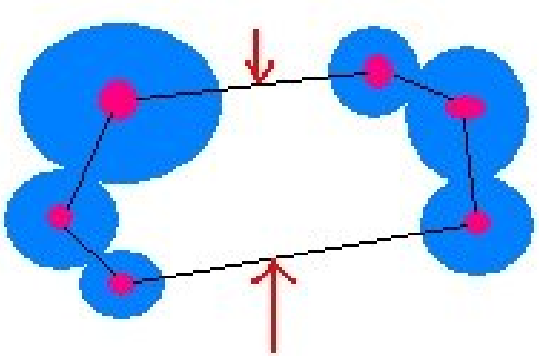


Figure 3: The red arrows display what the algorithm will overlook and the overall problem with the simple form of this method.

Linear programming and this approach is not without its drawbacks. As the problem sizes grow, so too, do the number of constraints, $O(n^2)$, that must be solved for. In addition to the growing constraint problem, we have to worry about the quality of the bounds solution produced. The six city example, shown in Figure 2, is an ideal example since all of the cities touch, but if we take a look at an example where we can't get all of cities to touch, the lower bound will be deceptively small since our bound of $Maximize(2r_1 + 2r_2 + 2r_3 + 2r_4 + 2r_5)$ won't take into account the distance the salesman must travel to overcome the gap. Refer to Figure 3

There are means to overcome this problem, but they are beyond the scope of this discussion.

0.4 Metaheuristics

In the last 20 years, a new kind of approximation algorithm has emerged which tries to combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space. These methods are nowadays commonly referred to as *metaheuristics*. The term *metaheuristic*, first introduced by Glover in 1986 [11], is best described as an iterative search strategy that guides the process over the search space in the hope of finding the optimal solution. This class of algorithms includes, but is not restricted to, Ant Colony Optimization (ACO), Evolutionary Computation (EC) including Genetic Algorithms (GA), Iterated Local

Search (ILS), Simulated Annealing (SA), and Tabu Search (TS). Generally, metaheuristics are not problem specific and contain mechanisms to avoid getting trapped on local optima. Up until recently there was no commonly accepted definition for the term *metaheuristic*, but in the last few years, some researchers in the field have tried to propose a definition (Refer to [4] for several other proposed definitions):

"Metaheuristics are typically high-level strategies which guide an underlying, more problem specific heuristic, to increase their performance. The main goal is to avoid the disadvantages of iterative improvement and, in particular, multiple descents by allowing the local search to escape from local optima. This is achieved by either allowing worsening moves or generating new starting solutions for the local search in a more "intelligent" way than just providing random initial solutions. Many of the methods can be interpreted as introducing a bias such that high quality solutions are produced quickly. This bias can be of various forms and can be cast as descent bias (based on the objective function), memory bias (based on previously made decisions) or experience bias (based on prior performance). Many of the metaheuristic approaches rely on probabilistic decisions made during the search. But, the main difference to pure random search is that in metaheuristic algorithms randomness is not used blindly but in an intelligent, biased form." [30].

Metaheuristic algorithms are best described by how they operate over the search space [4]:

- Nature-inspired *vs.* non-natured inspired
- Population-based *vs.* single-point search
- Dynamic *vs.* static objective functions
- One *vs.* various neighborhood structures
- Memory usage *vs.* memory-less methods

Algorithms and Theory

0.5 Genetic Algorithms

Genetic algorithms (GA) are population based stochastic search algorithms inspired by Darwin's Theory of Evolution and were first introduced by John Holland in the early 70's. At a high level, the GA begins with a randomly generated population of potential solutions. Each member of the population is represented by a DNA string, a *chromosome*, that is an encoding of the problem and search space. Members of the population are selected for recombination based on fitness scores and recombined to form a new population, i.e., the next generation. This process is repeated until some termination criteria are satisfied, e.g., number of generations, run time, or improvement of the best solution. The GA's population based approach gives it implicit parallel computational abilities and also allows for program level parallelism.

GAs can be classified by how they create the next generation, being either *generational* or *steady-state.* With a generational based GA, there are two separate populations: *parents* and *children.* The *parent* population is used to select members from for breeding, placing the resulting offspring into the *child* population. The generational GA delays the offspring's reproductive participation until the next generation. The main benefit of this type of GA is it is less likely to prematurely converge on a possibly inferior solution. The downside is that good individuals, those with high fitness scores, have to wait until the next generation before their genetic material is used. In short, this methodology delays the offspring's reproductive participation. A steady-state GA maintains one population where the offspring are placed back into the population with, or replacing, their parents. Because of this single population implementation, in steady-state GAs, the offspring's reproductive participation is immediate, but it is also more likely to prematurely converge on a solution than its generational counterpart.

There are two components to a genetic algorithm that are problem dependent:

- Representation of problem as (bit) strings
- Trial (solution) evaluation

Basics of a Genetic Algorithm

0.5.1 Encoding

GAs use an analog of "DNA" to encode the problems they are solving and their associated search spaces. The DNA can be binary, i.e., consist entirely of 0's and 1's, or something entirely different. Parameters and variables the GA will have to consider, optimize, or search have to be encoded in a format the GA can recombine, mutate, and evaluate. If the parameters are not discrete, then the encoding has to be sufficient enough that the GA doesn't loose any precision.

In the case of the TSP, the problem and search space was encoded as a permutation of the city ordering. If the final result is the string '1 2 3,' then the salesman would visit city 1 first, then city 2, then city 3, returning to city 1. The process of encoding the problem in the DNA is the most difficult as well as one of the most important aspects of GA design because your encoding defines your solution space and how your answer will look as well as how the GA will act.

Encoding is a trade-off between flexibility and search space. Choosing a more focused encoding scheme will yield faster results, but doing so can confine the GA to a smaller portion of the search space. Restricting the GA severely limits one of the GA's most attractive features: its ability to produce unique and clever solutions.

0.5.2 Evaluation

The evaluation phase of a GA involves the assignment of a numerical ranking to each member of the population. These values are referred to as the member's *fitness* score and are used in determining which individuals are chosen for incorporation into the next generation. Due to the prominence these values play in the overall operation, and ultimately success, of a GA, it is *extremely* important that the evaluation function is designed well. Efficiency is something else to consider since the ranking algorithm will be invoked for every member of GA every generation it is run!

Like the problem encoding phase of the GA, the evaluation phase also involves some trade-offs. Objectives of the evaluation phase include the creation of a mathematical function that can "grade" the DNA string both quickly and accurately since it is an operation the GA might have to perform over and over. For example, if the GA runs for 100,000 generations and has a population of 100, you can expect your fitness function to be called upwards of 10,000,000 times!

The evaluation strategy employed by my GA computes the total distance traveled by the salesman. Hence, a lower distance equates to a better score. This strategy has several advantages: (i) it is the simplest evaluation strategy for this problem and (ii) it is deterministic in terms of the evaluation provided for any given member. A more complicated evaluation strategy might look at the number of overlaps in the current path and add a small penalty for each one in addition to the total distance traveled. This evaluation strategy would award paths that don't backtrack in addition to being shorter, but due to the complexity of detecting an overlap, this evaluation algorithm could take a big hit in running time.

0.5.3 Parent Selection

Selection is the process of choosing the parents whose DNA will be incorporated into the next generation. Several paradigms exist including choosing from a pool of only the best parents, *elitism,* to choosing a random sampling, *tournament, roulette, and stochastic* selection algorithms. Elitism takes the view that you will get best results if you ignore a percentage of the worst performers while the others believe it's best to randomly sample the entire population. Neither approach is without its drawbacks. Elitism, improperly used, i.e., ignoring too large a portion of the worst members, will lead to quick and premature convergence while the others can ignore good genetic material in preference of inferior solutions, ultimately resulting in inferior solutions. Harem select, a method first introduced in this thesis is a hybrid operator that borrows from both the elitist camp and the inclusive camp.

All of the parent selection methods used are described in more detail in the following

paragraphs.

Tournament Selection Tournament selection randomly selects two individuals from the population and compares their fitness values. The one with the best fitness is allowed to mate, the loser is replaced with the offspring. Another tournament is used to select the other parent. Two parents are selected using two two-member tournaments.

Roulette Wheel Sampling Under Roulette Wheel Sampling, each member is assigned a slice of a roulette wheel where the size of the slice is proportional to the individual's normalized fitness. The wheel is spun up to N times where N is equal to the number of members in the population. The parent is then selected based on the cumulative sum of fitness. Two parents are selected using two roulette wheel samplings. Member replacement is linear, that is, the GA replaces members 0 and 1 with the children of the first two parents selected, then 2 and 3, and so on until the next generation is created.

Stochastic Remainder Stochastic remainder is a probabilistic selection method - similar to a weighted roulette wheel. Each individual is assigned as many copies as the integer part of its expected number of instances. Then, for the unoccupied slots in the next generation, a roulette wheel-like selection is carried out based on the residues of the expected number of instances of each individual.

Elitism Elitism's goal is to address the problem of loosing the individuals with the best fitness during the optimization phase of the GA. Often, this is done by preserving the best individuals from the old population and combining them with the best of the new population. This is in opposition to the other methods that replace the entire old population with individuals created by recombining them (recombination of the old population that is) [10].

The elitism algorithm employed by this GA allows a specified percentage of the best fit individuals to survive and reproduce, replacing the least fit individuals along the way. This percentage of the top individuals encompass the "breeding pool." For example, if the elitism rate was set to *.5*, then only the top 50% of the population would be considered for parent selection, with the bottom 50% being those individuals who will be replaced by the next generation.

To get a better idea of what is going on, let's look at an example of how this form of elitism is implemented. You can follow the *elitistGA* psuedocode below for a little more clarity. Let's assume the population size is 10 and we're using an elitism rate of 50%. After we generate the initial population, we evaluate, or score, all the members. After the evaluation phase, we sort the parent population ($Population_1$ in the pseudocode below) such that the best individual is in index 0 and worst individual is in the last index position. For this example of elitism, we will only replace the worst 5. Why only 5? Remember, we're using an elitism rate of 50% and the population size is 10. 50% of 10 is 5! Getting back to the example, we automatically copy over the top 50% of the population, or 5 in our case, to the new population ($Population_2$ in the pseudocode below). Once this is done, we perform the standard genetic operators of parent selection, recombination, and mutation to $Population_1$. The new members are placed into the 5 unfilled positions of the new population ($population_2$ in the elitistGA pseudocode). Now we sort the new population ($population_2$) to get the best individuals back to the top of the parent array. This is needed since the recombination and mutation operators could have created individuals that are of a higher fitness! The new population is sorted and we start all over again.

Since there is a lot of sorting, the process is optimized by storing all of the fitness values in an array so that these values don't have to be continually recomputed. On top of this, after the initial sort is performed, from there on out, the population arrays are usually close to an optimal sorting, so we want to choose a sorting algorithm that won't produce worst case results if the structures to sort are already sorted or pretty close. Another point to take note of is the fact that the copy of the top 50% of $Population_1$ to $Population_2$ happens only once - when the algorithm first starts up. After this, $Population_2$ will always contain the sorted population.

The elitism percentage is adjustable and can be set anywhere in the range of 0 to 1.0. If the rate is set to .25, the GA will only consider the top 25% of the population. To simplify the selection and replacement process, the population is sorted according to fitness. Once the GA knows where the lower boundary of parents to select from is, it performs roulette wheel sampling on the remaining members.

After performing a literature search, I couldn't find any references to elitism that operated in such a way, so I'm working on the assumption that it hasn't been done in a while or this is a new version of elitism.

Algorithm 0.5.1: ELITISTGA(*void*)

$randomlyCreate(Population_1)$

$score(Population_1)$

$sort(Population_1)$

comment: Copy over the portion of the population that the

comment: elitist rate specifies we maintain. These individuals

comment: are maintained in the upper-half of the population array

for $i \leftarrow 0$ **to** $(int)Population_2.length/elitistRate$

  $Population_2[i] \leftarrow Population_1[i]$

for $j \leftarrow 0$ **to** $totalGenerations$

  for $i \leftarrow (int)Population_2.length/elitistRate$ **to** $Population_2.length$

    comment: randomly select member between 1 and max allowed elitist pop

    $parent_1 \leftarrow Population_1[random(0, Population_1.length/elitistRate)$

    $parent_2 \leftarrow Population_1[random(0, Population_1.length/elitistRate)$

    $recombine(parent_1, parent_2, offspring_1, offspring_2)$

    $optionallyMutate(offspring_1)$

    $optionallyMutate(offspring_2)$

    $Population_2[i] \leftarrow offspring_1$

    $Population_2[i+1] \leftarrow offspring_2$

    $i \leftarrow i + 2$

  $sort(Population_2)$

  $Population_1 \leftarrow Population_2$

return

Harem Selection The harem selection technique comes in two distinct flavors, a purely elitist form and hybrid form. Both of these methods are modeled after the biological social

hierarchy also referred to as a harem and operate very similar to the biological construct that is their namesake. In biology, there is a dominant individual, normally the male in mammals, which has mating rights to a harem, or group, females. The harem selection method uses a similar setup - a dominant individual, i.e., the individual with the best overall fitness in the current generation, is crossed with the other individuals in the population.

Where the two forms differ are in how they select the members for recombination with the dominant individual - one uses roulette wheel sampling and the other uses elitism. From generation to generation, if no new member is "born" that has a higher fitness score, then the old dominant individual retains his post as the best and isn't replaced until a better individual comes along. If a new individual comes along with the same score as the best but has a different genetic sequence, then the old still retains his post. Swapping is an option to be tested since the new genetic makeup could blend better with the current population.

Looking at the pseudocode below, you will notice a great similarity between harem selection and elitism selection. Where they differ is minor, but very important. I've highlighted the differences to make them a little more apparent.

Algorithm 0.5.2: HAREMGA(*void*)

$randomlyCreate(Population_1)$

$score(Population_1)$

$sort(Population_1)$

comment: Index zero always contains the best known individual.

$Population_2[0] \leftarrow Population_1[0]$

comment: Copy over the portion of the population that the

comment: elitist rate specifies we maintain. These individuals

comment: are maintained in the upper-half of the population array.

for $i \leftarrow 1$ **to** $(int)Population_2.length/elitistRate$

$Population_2[i] \leftarrow Population_1[i]$ **for** $j \leftarrow 0$ **to** $totalGenerations$

for $i \leftarrow (int)Population_2.length/elitistRate$ **to** $Population_2.length$

comment: parent 1 should ALWAYS get the best individual

$parent_1 \leftarrow Population_1[0]$

comment: randomly select member between 1 and max allowed elitist pop.

if $type \equiv pureElitist$

$parent_2 \leftarrow Population_1[random(1, Population_1.length/elitistRate)$

else $parent_2 \leftarrow RouletteWheelSelect(Population_1)$

$recombine(parent_1, parent_2, offspring_1, offspring_2)$

$optionallyMutate(offspring_1)$;

$optionallyMutate(offspring_2)$;

$Population_2[i] \leftarrow offspring_1$

$Population_2[i+1] \leftarrow offspring_2$

$i \leftarrow i + 2$;

$sort(Population_2)$;

$Population_1 \leftarrow Population_2$

return

0.5.4 Mutation Operator

With some low probability, a portion of the new individuals will be slightly, randomly modified. For a bit-string representation, this amounts to inverting a few bits. The purpose of this operator is to maintain diversity within the population and inhibit premature convergence. Mutation alone induces a random walk through the search space while mutation and selection (without crossover) create a parallel, noise-tolerant, hill-climbing algorithm.

Since this particular GA is being applied to the TSP problem, a simple bit flip won't work since it will break other constraints within the problem, namely that the chromosome must remain a valid permutation. Otherwise, any decent GA will quickly discover the shortest path is one that continually visits the same city. Hence, if a position, or bit, is selected for mutation, a random number is drawn and this index is the one that will be swapped with the current position. There is no check against the random number being equal to the current position, so it is possible, albeit more unlikely the larger the problem, that a bit is selected for mutation and it does not occur. For example, if we're in the mutation phase of a 52 city problem and we're currently considering index 10 for mutation and randomly pick 34, then positions 10 and 34 get swapped. But if we're at index 10 and randomly pick 10, then for all practical purposes, no mutation actually took place! As the problem size, N, grows, the probability of this happening decreases proportionally, that is, $\frac{1}{N}$.

0.5.5 Crossover Operator

The crossover operator plays an important role in the overall operation of the GA since it is the technique that generates the exchange of information between the individuals in the population during the search. This exchange of information is often talked about as being explorative since crossover helps the GA discover promising areas within the search space by making large jumps.

0.5.5.1 Crossover vs. Mutation

It's a decades long debate on which one is better or necessary. The answer, or at least a general consensus, is that it depends on the problem, but in general, it is good to have both of them. They both play separate roles: crossover is explorative while mutation is exploitative. This allows for co-operation **and** competition between them. Crossover helps guide the GA over the entire search space while mutation creates random, small perturbations, staying in the area of the parent. While a mutation only GA is possible, a crossover only GA would not work! Why is this? Only the crossover operator can combine information from two parents, but only the mutation operator can introduce new information (alleles).

As a side note, using a GA to solve the TSP is an instance of "it depends on the problems" since it will work without mutation *if* the encoding scheme used is a permutation of the cities to be traveled. It works since every possible city is already present, the GA just has to find an optimal ordering. In this case, there is no need for an operator that introduces new information, only operators that re-order the existing information.

0.5.5.2 Goals

A problem commonly encountered when using a genetic algorithm to solve a problem is having the GA prematurely converge or getting stuck on a local optima. The GA's stochastic nature is suppose to help limit these problems, but in real applications, the results have generally been less than what a simple greedy algorithm can produce. Now take into account the amount of computational power you need to power a GA and they don't often look like the best path to take.

This thesis will do an overview of some of the most commonly employed techniques in the arena of genetic algorithms. In addition to this, I will present some new enhancements and hybrid techniques that will boost the overall performance upwards of 500%. Topics covered will include the most commonly used parent selection methods: *roulette* and *tournament* in comparison with lesser used selection techniques such as *stochastic uniform* and *elitism*. In

addition to these, I will present one new parent selection method that comes in two forms, *harem selection*. Hands down, harem selection out-performed the other selection methods with very few exceptions!

Other areas covered will be the most commonly used recombination algorithms as well as some not so commonly used ones. Partially Matched Crossover (PMX) and ordered crossover (OX) are permutation crossovers commonly employed by researchers using GAs. Uncommonly employed crossovers include merge order crossover (MOX) and edge assembly crossover (EAX). EAX is a TSP specific crossover that preserves some orderings and uses heuristics when deciding which edges to add.

To complement the parent selection and recombination techniques, I will also explore memetic algorithms - a hybrid technique. A memetic algorithm is a GA that replaces the mutation operator with a local search. Some other new concepts to be explored will be population seeding, adaptive mutation rate, restart, and hybrid forms of the GA encompassing various combinations of the previously mentioned techniques.

0.5.5.3 Genetic Algorithms and TSP

Many genetic algorithms have been described for solving the traveling salesman problem. Grefenstette [14] presented Heuristic Crossover (HEU), also called greedy crossover. HEU picks a random starting point and iteratively looks for the shortest edge in the two parents extending the tour from the most recent city. If a cycle is produced, the next city is randomly chosen and the algorithm moves on. More recently, Nagata and Kobayashi [26] have introduced Edge Recombination Crossover (EAX). EAX is a weak method needing a large population to obtain good results. It builds offspring by only using edges present in both parents. EAX constructs an edge table and then proceeds to greedily choose the next edge, making a random choice if conflicts or cycles arise. Julstrom [19] applied a very greedy crossover similar to Grefenstette, while others [9], [24], [25] have applied hybrid-genetic operators that reported near optimal results on TSPLIB's [20] 532 city problem.

More recently, [32] and [17] have employed genetic algorithms that replace and/or supplement the mutation operator with a simple local search such as 2-opt. These implementations are really memetic algorithms [23], [5] and have reported optimal or near optimal results on many of TSPLIB's smaller libraries!

0.5.5.4 Methods Employed

The genetic algorithm used allows duplicate members and is a generation based GA as opposed to a steady-state GA. See the pseudo-code below for a generalized version. Other researchers like Watson, et. al., [31] didn't allow duplicates in their GA and used a method they coined *iterative child generation*, or ICG, to produce up to 100 children, looking for offspring with a better fitness score than their parents. They reasoned disallowing duplicates preserved genetic diversity since the symmetric TSP only has $(N^2 - N)/2$ edges to choose from and a population must be of size $(N - 1)/2$ to encounter each edge just once. Gottlieb [13] showed that the steady state model of children competing against the parents produces a high selection pressure that continually produces increasing fitness values, but can lead to a premature convergence if some individuals start to dominate the population. Implementing Watson's, et. al. idea of ICG might prove useful to get around or delay the premature convergence problem of steady-state GA's, but will cost the GA in the total number of required fitness evaluations.

Algorithm 0.5.3: GENERATIONALGA(*void*)

t = 0

$initialize(Population_1)$

$evaluateFitness(Population_1)$

while $t < generations$

  Create empty $Population_2$

  for $i \leftarrow 0$ **to** $(PopulationSize - 1)$

    $tempIndividual_1 = \text{selection}(Population_1)$

    $tempIndividual_2 = \text{recombination}(Population_1)$

    $P[i] = \text{mutation}(tempIndividual_1)$

    $P[i+1] = \text{mutation}(tempIndividual_2)$

    $evaluate(Population_2)$

    $swap(Population_1, Population_2)$

  t = t + 1

return

Genetic algorithms are famous for prematurely converging on local optima when it comes to many problems. To avoid the problem of premature convergence, [1] used a Self-Organizing Maps (neural networks) to mine the past results of the GA and use these to guide the future direction. I employed several new and old methods to avoid getting trapped on local minima because of premature convergence. These methods include adaptive mutation, restart, local search, and larger population instances. These are all discussed in more detail in the following paragraphs.

Six methods of parent selection were implemented and tested: *tournament select*, *roulette select*, *stochastic remainder*, *elitism*, and two versions of a new parent selection the author refers to as *harem select*. Again, the GA also uses several methods of combinatorial recombination: Partially Matched Crossover (PMX), Ordered Crossover (OX), Merge-ordered crossover (MOX), and edge assembly crossover (EAX). Having so many selection and crossover techniques allows

for a side-by-side comparison to see if there are techniques that are superior to the others.

The parameters of a typical GA are: population size, the maximum number of generations to run *or* the time to run, crossover probability and mutation probability. All these parameters were set as defined in table 2 and remained constant throughout testing. The values were chosen since they are, or are close to, the most commonly used values in the published literature.

Table 2: GA Parameter Configuration Table

Edge Table	
Parameter	Value
Mutation Rate	.5%
Crossover Rate	60%
Population Size	250 & 500
Generations	10000 & 50000
Tournament Size	2
Elitism Rate	50%
Seed Count	1

The GA was run 2 times with 2 different stopping parameters - 10,000 and 50,000 generations. This was done to see if allowing the GA more time would make any difference in its overall performance. The number of generations was the GA's only termination criteria. This is a rather simplistic method when compared to others available such as meeting a performance goal (i.e., finding a solution within 1% of known optimal) or ending when the GA stopped improving the solution from generation to generation. There is a downside and upside to each of the methods just mentioned. Using a fixed number of generations is very simple and straight forward to implement and guarantees the GA will end without any tricky logic, but it can have the GA run longer than absolutely necessary. A relevant example would be the GA stopped improving the solution after 2,000 generations but was forced to run for 10,000. Forcing the GA to achieve a certain goal before termination doesn't guarantee the GA will ever terminate and requires a secondary check to determine when to give up. This too has the potential downside of running the GA longer than necessary and is more complicated to implement a secondary check alongside the primary goal. Ending the GA after so many generations of non-improvement ignores the random nature of a GA and its ability to climb out of local optima, but it can be the

most efficient in that it ends the GA at the first sign the GA has converged on a local optima.

When applied to the TSP problem, one of the most important configuration parameters that a GA can have is an appropriate population size that will encompass all possible edge combinations. Because of memory and time limitations, obviously this isn't always feasible, especially as the problem instances grow very large. With this in mind, the GA was tested using two population size configurations: 250 and 500 members.

Mutation is performed by swapping two positions since this will maintain the constraints set forth by the problem. Specifically, since the encoding used is a permutation of the tour, flipping bits won't work since it could produce tours that are no longer permutations. Otherwise, the GA would quickly figure out that the shortest tour is a tour that only visits the same city - a tour of zero distance!

Fitness is given by the Euclidean distance of the tour, the shorter the distance, the better the fitness. The Euclidean distance is calculated by equation 1.

$$d_{ij} = (int)(\sqrt{(x_i - y_i)^2 + (x_j - y_j)^2} + .05) \tag{1}$$

The GA also employed two new techniques - *population seeding* and *adaptive mutation.* These techniques were compared by running the GA using the new techniques and without using them to easily compare what their use would gain us. The GA was run once with seeding and adaptive mutation rate turned off, with only seeding turned on, with only adaptive mutation rate turned on, and with both adaptive mutation rate and seeding turned on.

0.5.6 Population Seeding

A new method this GA employed was the concept of seeding the population. A cornerstone of genetic algorithms is the random initial population - a GA starts off with a randomly generated population of potential solutions. The hope is that from this random genetic material will rise a very high quality solution, with the help of the genetic operators, of course!

But what if someone were to brake this rule and "seeded" the population with a member

or two that were known to be of a higher initial fitness? Though neither proven nor tested, logically it made sense that the GA would prematurely converge, or fail all together to even move from the seeded members if a good deal of its initial population weren't random. The reasoning behind this is the GA would lack the genetic diversity needed to properly operate. Keep in mind, genetic algorithms are not optimization algorithms, but stochastic search algorithms. With this constraint in mind, the seed count was restricted to only one individual. Julstrom [18] experimented with a hybrid GA for the Rectilinear Steiner Problem that seeded the initial population with organisms that were of a higher initial fitness. Julstrom's work has neither been reproduced nor has this method been applied to other problems. The seeded solution was created using the *ConstructRandomizedGreedySolution()* method used by the GRASP algorithm with $\alpha = 5$ (See the section on GRASP to read about the significance of α).

0.5.7 Adaptive Mutation Rate

As a GA progresses, the goal is to have the overall fitness of the parent population increase from one generation to the next. Taking a closer look at what this really means, we would expect that the ordering of the genes in each of the individuals is to be arranged in such a way that each succeeding allele is in a position that increases the individual's overall fitness, or is at least closer to an optimal ordering. If we let the GA continually make large positional jumps for the alleles via the mutation operator at the later stages of parent development, will this have an adverse affect on the overall success of this strategy? Put another way, can we expect better results if we limit the distance an allele is moved via mutation as the GA progresses? This is a question that will be investigated for the GA portion of this thesis.

As the number of generations increases, the algorithm only allows swaps that are closer and closer to the index we're currently located at - the thinking is as the GA progresses, larger swaps will be fairly detrimental since it should be closer to a decent solution (optimal ordering), hence a closer swap will be more likely to yield an improvement, or at least less likely to destroy a good solution. For demonstration sake, lets assume we have a 20 bit chromosome and the

GA is set to run for 100 generations. For the first 25% of the generations, or $gen = 0$ through $gen = 24$, the GA will allow the mutation operator to move an allele any distance and in any direction from its current position in the DNA string, please refer to table 3. For the second 25% of the generations, or $gen = 25$ through $gen = 49$, the mutation operator is restricted to an index range of 75% of the total distance - 35% in either direction, please refer to table 4. For our example GA, if the mutation operator has chosen index 10 for mutation, it is restricted to a move up to index 14 or down to index 6, but no further in either direction. For the third quarter, $gen = 50$ through $gen = 74$, and the last quarter, $gen = 75$ through $gen = 99$, the mutation operator is restricted to 25% and 12% of the chromosome length respectively, please refer to table 5 and table 6.

Table 3: Chromosome with mutation chosen for index 10, $gen = 0 - 24$

0	1	2	3	4	5	6	7	8	9	10 M	11	12	13	14	15	16	17	18	19

Table 4: Chromosome with mutation chosen for index 10, $gen = 25 - 49$

0	1	2	3	4	5	6	7	8	9	10 M	11	12	13	14	15	16	17	18	19

Table 5: Chromosome with mutation chosen for index 10, $gen = 50 - 74$

0	1	2	3	4	5	6	7	8	9	10 M	11	12	13	14	15	16	17	18	19

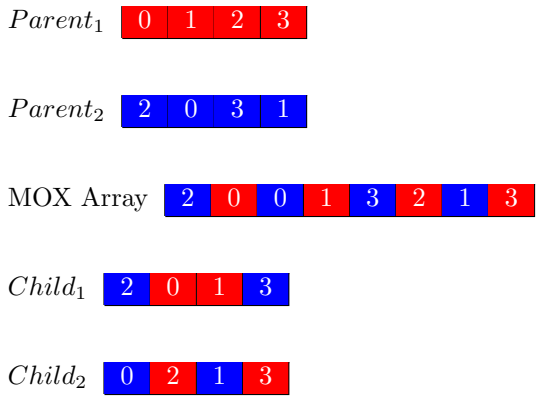
Table 6: Chromosome with mutation chosen for index 10, $gen = 75 - 99$

0	1	2	3	4	5	6	7	8	9	10 M	11	12	13	14	15	16	17	18	19

0.5.8 Crossovers: PMX, OX, MOX, & EAX

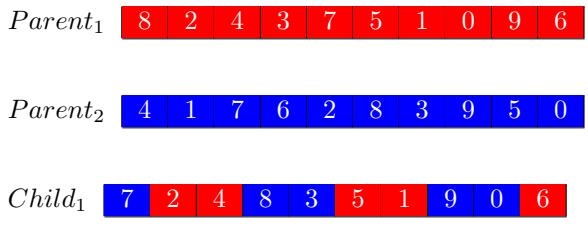
Merge Order Crossover Merge order crossover (MOX) [2] is likened to a rifle shuffle of playing cards - you merge the parents and grab the first occurrences of each value. MOX

preserves the order of cities when applied to TSP. **Note:** The following example uses a perfect merge. The real implementation will usually grab several alleles at a time to "merge" into the intermediary array. See below:



Each have the benefit of preserving the ordering of the alleles, i.e., the order cities are traveled. As the GA starts to find paths that yield optimal results in deceptive terrains, the GA will be less likely to destroy the ordering with crossovers like MOX, PMX, and OX.

Ordered Crossover Ordered Crossover (OX), first presented by [12], is a commonly used order preserving crossover that preserves some orderings as well as the absolute positions of alleles. The OX operator is shown in the arrays below: $parent_1$, $parent_2$, & $child_1$. Essentially what is happening is we're randomly picking about half the elements in $Parent_1$ and copying them over to $Child_1$, preserving their original positions. The remaining elements are taken from $Parent_2$ and copied to $Child_1$, preserving their order within $Parent_2$. In this example, we choose 2, 4, 1, 5, & 6 from parent 1 ($Parent_1$). The remaining elements, 0, 3, 7, 8, & 9, are taken from parent 2 ($Parent_2$).



Partially Matched Crossover Partially Matched Crossover (PMX) preserves the absolute positions of the alleles in the parents when creating the children compared to OX that

preserves some orderings as well as absolute positions of alleles. In it's simplest form, PMX randomly picks a position in the two parents and interchanges the elements found there. Refer to the arrays below labeled: $Parent_1$, $Parent_2$, $Child_1$, & $Child_2$. The arrays labeled $Parent_1$ and $Parent_2$ show the parents before PMX. The red cell is the randomly chosen index whose members we wish to swap - 5 for $Parent_1$ and 8 for $Parent_2$. The arrays labeled $Child_1$ and $Child_2$ display what the new members will look like.

$Parent_1$	8	2	4	3	7	5	1	0	9	6
$Parent_2$	4	1	7	6	2	8	3	9	5	0
$Child_1$	5	2	4	3	7	8	1	0	9	6
$Child_2$	4	1	7	6	2	5	3	9	8	0

Edge Assembly Crossover Edge Assembly Crossover (EAX) builds offspring using only the edges present in both parents, and because of this, this method works better with larger parent populations since there is a better chance of more unique edge combinations being present. EAX first builds an edge list table from the two parents. Alleles are then chosen one at a time by choosing the allele with the smallest number of edges between the alleles connected with the current one. If all are of equal length, a random edge is selected. After an allele has been chosen, it is removed from the edge table and the connected alleles are considered for the next edge. Refer to Table 7. This is repeated iteratively until all alleles have been incorporated in the offspring [26].

$Parent_1$	0	1	2	3	4	5	6	7	8
$Parent_2$	8	0	6	1	4	2	3	5	7
$Child_1$	8	7	6	5	3	4	2	1	0

0.5.9 Random Restart

While the concept of random restart is not new, it is when discussed in the same sentence as genetic algorithms. How it would be theoretically applied would be to randomly apply it after

so many generations. Once it was to be applied, the GA would randomly select a portion of the lowest performers, dump them, and randomly regenerate them. In practice, it was applied as follows: the algorithm would wait until 75% of the time allowed elapsed and dump the lowest 2/3 of the population. These members were regenerated using the exact same methods used to randomly generate the initial population. This method would be best described as restart.

In local search algorithms, the goal of a random restart is to get the algorithm "unstuck." Unguided algorithms have a tendency to get stuck on local optima, randomly restarting the search is an explicit strategy to move the search onto yet to be explored terrains of the search space.

0.5.10 Genetic Algorithms with Local Search

The GA was tested using a local search instead of the mutation operator. The main goal of the mutation operator is to maintain diversity. Using a local search in place of the mutation operator will have a similar affect since it will look at solutions in the local neighborhood of the current solution, hopefully replacing them with something better. Since the GA is already a computationally expensive algorithm, we don't want to use a local search that is also expensive. With this constraint in mind, **2-Opt** was chosen because it is simple, quick, and has been proven to produce excellent results on the TSP problem.

Table 7: Edge table produced by parent 1 and parent 2

Edge Table	
City	Connected To
0	1 8 6
1	0 2 6 4
2	1 3 4
3	2 4 5
4	3 5 1 2
5	4 6 3 7
6	5 7 0 1
7	6 8 5
8	7 0

0.5.11 Hybrid GA Implementations

In the end, the GA was tested using all the techniques listed above: adaptive mutation rate, random restart, local search (2-opt), and the various combinations of parent selection and crossover algorithms.

0.6 Greedy Randomized Adaptive Search Procedure

Greedy Randomized Adaptive Search Procedure, or GRASP, is a simple two phase iterative algorithm that combines heuristics and a local search. Phase one consists of solution construction while phase two is solution improvement. [4]

0.6.1 Construction Phase

The construction phase of GRASP is essentially a pseudo-greedy algorithm. During this phase, a feasible solution is constructed one element at a time. At each successive step, one new element is added by picking it at random from a list of potential candidates. A heuristic is used to rank and score each element at each successive step. The candidate list, also referred to as the *restricted candidate list* (RCL), is composed of the remaining best α elements - those elements that have yet to be chosen for the solution. Each element's fitness score is updated after every iteration, thus the score of one element can change from iteration to iteration depending on the algorithm used for determining fitness. For example, distance would be the metric of choice for the TSP, so an element's fitness score will change as its distance from the city most recently added to the tour changes. This continuous updating of the element's fitness is what makes this algorithm adaptive and the random choice technique allows for different solutions after every construction phase. Repeated applications of the construction procedure yields diverse starting solutions for the local search to improve upon. The second psuedocode listing describes the basic construction phase. Refer to the following example for a better picture of how the construction phase works.

Let's assume a 100 city map, $1, 2, 3, \ldots, 99, 100$ and an alpha value of 5 ($\alpha = 5$). If we just

start with city 1, the partial solution will look like: | 1 | To generate the RCL, we loop through the remaining 99 cities and find the closest five cities to city 1. After looping through the remaining cities, let's assume the RCL looks like: | 5 | 13 | 19 | 89 | 91 | We randomly select an entry from the RCL. Let's assume entry index two: city 19. Now the partial solution will look like: | 1 | 19 | Repeating the previous algorithm, we loop through the remaining 98 elements and find the closest five cities to city 19. After looping through these remaining cities, let's assume our RCL looks like: | 5 | 17 | 42 | 77 | 91 | Again, one of these five cities is randomly selected and added to the partial solution and the whole process is repeated again.

What might not be fully evident from the pseudo-code is that the parameter α controls how greedy the construction phase is. Setting α to one will turn the construction phase into a pure greedy algorithm while setting α to the maximum allowed size, or even very close, will turn the construction phase into a purely random algorithm. A quick note, the maximum allowed size is the total number of elements needed to construct one solution, so on a ten city TSP map, setting $\alpha = 10$ will generate a purely random solution.

Due to the random nature of the construction phase, the solutions generated are not guaranteed to be optimal, or even close, hence the need for the local search phase. Most local search algorithms work in an iterative fashion. They successively replace the current solution with a better solution located in the neighborhood of the current solution. The local search will terminates when it can find no better solution in the neighborhood.

A solution s is said to be locally optimal in the neighborhood N if there is no better solution in N(s). The key to success for a local search algorithm consists of the suitable choice of a neighborhood structure, efficient neighborhood search techniques, and the starting solution. Depending on the quality of the starting solution, a local search algorithm can require an exponential amount of time to finish. Hence, starting with a higher quality initial solution can improve the performance of the local search. These ideas are the inspiration behind GRASP - the GRASP solutions are usually significantly better than any single solution obtained from a random starting point. The local search algorithm employed is the 2-Opt algorithm described

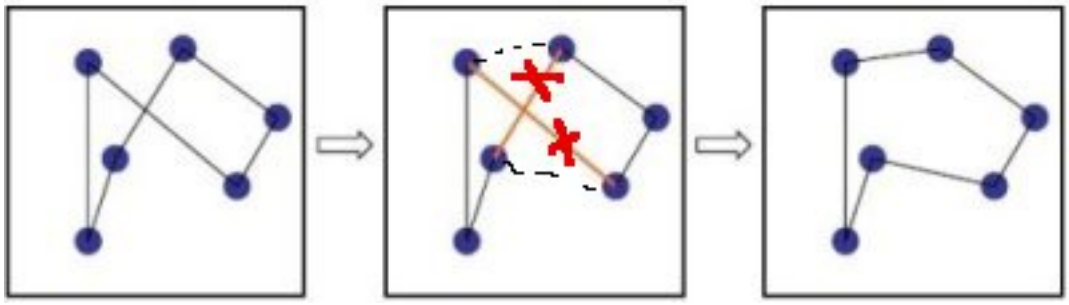


Figure 4: Example of 2-opt: The edges marked with 'x' are deleted and the dotted edges added.

by the pseudo-code below.

0.6.2 Local Search Phase

The second phase of GRASP uses a local search algorithm to improve the solution found in phase one. This local search algorithm can be simple like iterative improvement to more advanced and complex like simulated annealing or tabu search. For this implementation, I choose to go with a rather simple but powerful local search algorithm - 2-Opt.

2-Opt Algorithm 2-opt is a well-known local search algorithm for the Traveling Salesman Problem. A 2-opt exchange deletes two edges and replaces them with two new edges that do not break the tour into two disjoint cycles. Once the algorithm chooses the two edges to delete, there is no choice about which edges to add, there is only one way the edges can be added to create a valid tour. Figure 4 shows an example of this.

The 2-opt algorithm repeatedly looks for 2-opt exchanges that decrease the overall length of the tour. A 2-opt move decreases the length of a tour when the sum of the lengths of the two new edges is less than the sum of the lengths of the two deleted edges.

The 2-opt algorithm can be readily applied to large problem instances of the traveling salesman problem, but it's not without serious drawbacks. Like most search algorithms, it can easily become stuck on a local minimum. There are several methods that can help avoid this, one being simulated annealing with a slow cooling schedule. Another is called random restarts. As you recall, random restarting was the method employed by the GA. Simulated annealing

was the local search algorithm originally employed, but it required inordinate amounts of time and didn't produce results all too different from the pure SA algorithm. In fact, the difference in performance between SA alone and GRASP/SA was usually less than 5%! Since there was almost no difference in performance between SA and GRASP/SA, it was decided that a different local search algorithm should be employed.

Algorithm 0.6.1: GRASP($void$)

while termination conditions not met

$s \leftarrow ConstructGreedyRandomizedSolution()$

$ApplyLocalSearch(s)$

$MemorizeBestSolution()$

return

Algorithm 0.6.2: ConstructGreedyRandomizedSolution($void$)

comment: s denotes a partial solution in this case

$s \leftarrow null$

comment: definition of the RCL length

$\alpha \leftarrow DetermineCandidateListLength()$

while solution not complete

$RCL_\alpha \leftarrow GenerateRestrictedCandidateList(s)$

$x \leftarrow SelectElementAtRandom(RCL_\alpha)$

$s \leftarrow s \cup x$

comment: update of the heuristic values (see text)

$UpdateGreedyFunction(s)$

return

Algorithm 0.6.3: 2OPT($tour2Improve$)

while further improvements OR a specified number of iterations

$modifiedTour \leftarrow apply2optExchange(tour2Improve)$

if $len(modifiedTour) < len(tour2Improve)$

$tour2Improve \leftarrow modifiedTour$

return

0.7 Greedy Search

Greedy search is a problem specific neighborhood search. The algorithm looks in the immediate neighborhood of the current node and uses logic specific to the problem in deciding which node is added next. The greatest influence on the effectiveness and efficiency of the search is a tradeoff between the size of the problem and solution quality vs. the time allocated to search.

The neighborhood examination mechanism, or the order in which the algorithm searches the neighborhood, is extremely simple - the algorithm begins by picking the city at index zero (you could also randomly pick the starting city) as the starting point and iteratively choosing the next closest city to extend the tour. This is repeated until there are no more cities left to visit. See the pseudocode labeled **GreedySearch**.

Local searches have some disadvantages, the most notable being the short-sightedness of the search. With a problem like the TSP, there is an exponential increase in the number of local optima and a short-sighted search is more likely to get trapped in one of these local optima.

The main benefit of this algorithm is it's extremely fast and efficient - $\mathcal{O}\ (f(n))$, where n =the number of cities. In addition to being efficient, it is able to produce tours that are decent in terms of the overall distance traveled. But with the extreme simplicity comes a price: even on maps as small as 52 cities this algorithm will produce tours that back-track and cross paths with previous legs of the journey.

This thesis uses the *greedy algorithm* more as a control algorithm. Its results, both distance and timing, can be compared against the results of the more complex heuristic algorithms.

Because of its extreme simplicity, the algorithm should take only seconds even on the largest of problems to render an answer. Additionally, since it does use a simple heuristic, we can be assured the solution produced will generally be better than a randomly constructed one. This should allow us to gauge how much we gain by the added complexity inherent in the metaheuristic algorithms being studied.

Algorithm 0.7.1: GREEDYSEARCH(*void*)

$tour_0 \leftarrow cityList[0]$

$removecityList[0]$

for $i \leftarrow 1$ **to** n

  $d = distance(tour_i, cityList_0)$

  for $j \leftarrow 1$ **to** $cityList.length$

    if $distance(tour[i], cityList[j]) < d$

      $best \leftarrow j$

      $d = distance(tour[i], cityList[j])$

  $tour_i \leftarrow cityList[j]$

  $removecityList[j]$

return

0.8 Reactive Tabu Search

0.8.1 Background

The basic concept of Tabu Search (TS) as described by F. Glover [11] "is a meta-heuristic superimposed on another heuristic." TS explores the solution space by moving at each iteration from a solution s to the best solution in a subset of its neighborhood *N(s)*. According to Glover [11], this method was in part motivated by the observation that human behavior can appear to operate with elements of randomness that leads to inconsistent and sometimes sporadic behavior given similar circumstances. This resulting tendency to deviate from a charted course could be

regretted as a source of error but could also prove to be a source of gain. But unlike (some) human behavior, the tabu algorithm does not choose new courses randomly, instead the search proceeds accepting a new and possibly worse solution only if it is to avoid a path already investigated. This insures new regions of the search space will be continually investigated while avoiding local minima and ultimately potentially settling on a global maxima.

With TS, the main concept is to forbid the search to move to points already visited in the search space, at least for the next few iterations. These "tabu" moves help the search from getting trapped in local minima by allowing the algorithm to temporarily accept new, inferior solutions in an attempt to avoid paths already investigated. Tabu Search has traditionally been applied to combinatorial optimization problems, e.g., scheduling, routing, traveling salesman.

0.8.2 Implementation

Reactive Tabu Search (RTS) takes TS's concept of "tabu" moves, or the temporary prohibition of moves to points previously visited, one step further. With RTS, the "prohibition period" T is determined via a "reactive" feedback mechanism throughout the search. T starts at one, only preventing a return to the previous configuration, and is only increased when it is evident that diversification is needed - the repetition of a previously visited configuration. T is decreased when this evidence disappears, i.e., the algorithm finds unvisited configurations [3].

To avoid retracing the steps taken, all configurations found during the search are stored in memory in one or more tabu lists. The original intent of these lists was not to prevent a previous move from being repeated, but rather to insure it was not reversed. These lists are historical in nature and form the tabu search memory. After a move is executed, the algorithm checks whether the current configuration has already been found and reacts accordingly. Memory access and storage is performed by hashing or digital-tree techniques because of their constant time access properties. For non-trivial tasks, the overhead caused by the use of memory is negligible.

The first reactive mechanism is not enough to guarantee that the search trajectory does not get trapped in a limited region of the search space, thus, robustness requires a second

"escape" mechanism. This second escape mechanism is triggered when too many configurations are repeated too often. In addition, even if "limit cycles" (endless cyclic repetitions of a given set of configurations) are avoided, the first reactive mechanism is not sufficient to guarantee that the search trajectory is not confined in a limited region of the search space. A "chaotic trapping" of the trajectory in a limited portion of the search space is still possible (the analogy is with chaotic attractors of dynamical systems, where the trajectory is confined in a limited portion of the space, although a limit cycle is not present). For both reasons, to increase the robustness of the algorithm a second more radical diversification step (escape) is needed. The escape phase is triggered when too many configurations are repeated too often. A simple escape consists of a number of random steps executed starting from the current configuration (possibly with a bias toward steps that bring the trajectory away from the current search region) [3].

Some open problems of TS are [3]:

- The determination of an appropriate "prohibition period" for a given task (RTS addresses this with its self-adjusted prohibition period)
- The adoption of minimal computational complexity algorithms for using memory
- The robustness of the technique for a wide range of different problems.

Tabu Search is still actively researched and continues to evolve and improve. Most of the newly proposed tabu searches use specialized diversification strategies to guide the search.

Algorithm 0.8.1: TABU(*void*)

$s \leftarrow generateInitialSolution()$

$initializeTabuLists(TL[1], \ldots, TL[r])$

$k \leftarrow 0$

while termination conditions not met

$allowedSets(s, k) \leftarrow s'inN$

$s' \leftarrow chooseBestOf(allowedSet(s, k))$

comment: Update Tabu List And Aspiration Conditions

$k \leftarrow (k + 1)$

return

0.9 Simulated Annealing

0.9.1 Background

Simulated annealing (SA) is a probabilistic optimization technique analogous to the annealing process of metals which assume a low energy configuration when slowly cooled from high temperatures. Though notoriously slow, SA is well suited to CO problems with its main strength lying in its statistical guarantee of global minimization.

SA, considered to be the oldest among the metaheuristic algorithms, was one of the first algorithms to have an explicit strategy to avoid getting trapped in local optima. The algorithm's roots lie in statistical mechanics (Metropolis algorithm, 1958) and was first presented as a search algorithm for CO problems by Kirkpatrick in 1983 [21]. The basic idea is to allow moves that will result in a solution of lower quality than the current best solution. These "uphill moves" are used to escape getting trapped in local optima and are made with a probability that decreases as the search progresses.

0.9.2 Implementation

There are three major components to a SA implementation [27], [28]:

- Problem Representation
  - A means to represent the solution space
  - Function to measure the quality of any given solution
- Transition mechanism - a simple transition mechanism to slightly modify the current solution
- Annealing Schedule
  - Initial system temp: typically $t_1 = \infty$ (infinity) or a high value
  - Temperature decrement function: typically $t_k = \alpha \cdot t_{k-1}$, where $0.8 \leq \alpha \leq 0.99$
  - Number of iterations between temperature change - typically between 100 to 1000 iterations
  - Acceptance criteria
  - Stopping criteria.

0.9.3 The Annealing Schedule

An essential feature of the SA algorithm is the annealing schedule or cooling schedule. During the entire time the algorithm runs the temperature parameter, T, is gradually reduced. Initially, T is set to a high value or infinity (∞) and is decreased at each step according to the annealing schedule which is usually specified by the user, but must end with $T = 0$ at or close to the end of the allotted time budget. In this way, the system is expected to initially wander over a broad region of the search space containing good solutions and ignoring small features of the energy function. As the temperature cools, the system then drifts towards low-energy regions that become narrower and narrower, finally making only downhill moves via the steepest descent heuristic.

0.9.4 SA Basics

Simulated annealing starts by generating an initial solution, either randomly or heuristically (e.g., greedy algorithm), and initializing the temperature parameter T. The algorithm will then continually repeat the following until either the acceptance criteria *or* the stopping criteria is met: A solution *s'* from the neighborhood *N(s)* of the solution s is randomly sampled. SA always accepts moves that *decrease* the value of the objective function. As previously stated, moves that *increase* the value of the objective function are accepted with a probability that decreases as the algorithm progresses. So *s'* replaces s if $f(s') < f(s)$ or, in case $f(s') \geq f(s)$, with a probability which is a function of T and $f(s') - f(s)$. The probability is generally computed following the Boltzmann distribution, $e^{\Delta E/T}$, where ΔE is the difference in score between the current state and the chosen move, or $f(s') - f(s)$ [4].

In more abstract terms, the temperature T is decreased throughout the search process. Hence, at the beginning of the search, the probability of accepting uphill moves is high and gradually decreases as the search progresses, eventually converging to a simple iterative improvement algorithm. Regarding the search process, this means that the algorithm is the result of two combined strategies: random walk and iterative improvement. In the first phase of the search, the bias toward improvement is low and it permits the exploration of the entire search space. This erratic component is slowly decreased throughout the life of the run, thus leading the search to converge to a minimum. The minimum may be global depending on the cooling schedule. The probability of accepting uphill moves is controlled by two factors: the difference of the objective functions and the temperature. On the one hand, at a fixed temperature, the higher the difference $f(s') - f(s)$, the lower the probability to accept a move from s to *s'*. On the other hand, the higher T, the higher the probability of uphill moves [4], [27], [28].

Configuring SA involves a trade-off between the desire for a high quality solution and restricting computation time. The chance of getting a good solution can be increased by slowing down the cooling schedule, but by doing so, the algorithm will require more time to run. In

order to effectively use simulated annealing, a proper cooling schedule that will find a solution of high enough quality without taking too much time must be found.

0.9.5 Advantages

- Given some assumptions on the cooling schedule and certain restrictions, SA has been proven to converge to the optimum solution of a problem.
- An easy implementation of the algorithm makes it very easy to adapt a local search method to a simulated annealing algorithm.

0.9.6 Disadvantages

- Notoriously slow - although it is proven to converge to the optimum, it converges in infinite time.
- Since it requires slow cooling, the algorithm is usually not faster than its contemporaries.

0.9.7 Pseudocode

The pseudocode below describes the basic simulated annealing algorithm. The algorithm starts off with the while loop picking a random move in the neighborhood of the current solution. If the move is better than the current, it is always accepted, otherwise it makes the move with a probability that decreases as explained above.

Algorithm 0.9.1: s(←)

generateInitialSolution()

$T \leftarrow T[0]$

while termination conditions not met

$s' \leftarrow pickAtRandom(N(s))$

if $f(s'i) < f(s)$

$s \leftarrow s'$

else Accept s' as new solution with probability p(T, s', s)

$update(T)$

return

Testing Results and Conclusions

0.10 Research and the Traveling Salesman Problem

Algorithms used to solve the TSP have become ever more complicated and sophisticated since Dantzig, et. al. solved a 49 city problem in 1954 [8]. As evidence of this, the problem sizes researchers are solving have grown from 120 cities in 1980 [15] to 2,392 cities in 1991 [29] on up to 13,509 [7] cities in 1997. Since then, Applegate, Bixby, Chvatal, and Cook have put forth solutions to problems that are much larger than the 13,509 city map of the USA (all cities with a population over 500) [7].

Applegate, et. al., report times from 3.3 seconds for the gr120 problem instance using 1 node to approximately 10 years (collectively) for the usa13509 problem using a distributed network of over 9,500 nodes. Some of the larger problems they have tackled took even longer and required bigger networks to solve [7]. While their results are excellent, they're implementation restricts itself to but a minority of the population that may want to solve these types of problems since they require such large, dedicated networks and very sophisticated algorithms. Remember, their code has to not only be geared to solve the problem as efficiently as possible, but it also has to take into account large amounts of network traffic and the ability to recover if nodes

should disappear.

This large body of research and well documented results is what helps to make the TSP problem an excellent one to use as a test bed for new ideas. There are plenty of benchmarks in terms of both speed and performance to measure against. The TSPLIB is an online repository that hosts maps for different types of TSPs, e.g., symmetric, asymmetric, Euclidean 2D & 3D, etc. As well as hosting the problem instances, it also contains a wealth of knowledge and facts as well as the known bests for each of the problem instances it lists. TSPLIB can be found at http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.

0.10.1 Genetic Algorithms and the TSP

Genetic Algorithms are possibly one of the most over-used technologies in the metaheuristics field for the performance they deliver. So why then, is the research into such algorithms as GAs or genetic programming still so active despite the (overall) negative results encountered by most researchers? Think about it for a minute ... you wouldn't enter a Yugo into a NASCAR race (no offense to all of the Yugo owners out there)! All that being said and considered, GAs stick around for several main reasons, some technical, some not so.

One of the main reasons I believe they're so popular is because most everyone enjoys the idea of "playing God" just a little bit! You write this piece of code that "grows" and adapts. Like kids in a biological sense, you get to watch your creations actually learn and solve problems in ways you yourself might not even think of. Aside from being irresistible for most of us, it also has a fascinating aspect to it. In addition to this, GAs are much like the automobile - they have been refined and improved significantly since the day they were first introduced by John Holland. New techniques like co-evolution, speciation, island models, improved encoding and fitness rating, parent selection schemes like the new ones introduced in this thesis, as well as new crossover methods and other techniques not mentioned help in reducing the disadvantages of GAs of past and help move them towards being a tool that can surpass linear programming when one does not have access to the needed resources.

0.11 Testing Setup

All of the tests that were run were done so on a dual 2.2 GHz Opteron machine with 2 Gigabytes of RAM. The OS used was Suse Linux 10.0. None of the core algorithms used threading, so they used only one processor at a time. Timing results were obtained by taking a snapshot of the milliseconds before the test started and after the test ended and converting the difference to a clock value, so the timing values will also reflect OS activities such as task preemption as opposed to pure algorithm timing.

All distance values were calculated using the TSPLIB rule for the symmetric Euclidean 2-D problem. Its pseudo-code is listed below. Another thing to note are the negative values -48.39 in Table 8 and -0.54, -46.45 both from Table 11 are a result of out-of-date values from the TSPLIB site [20].

Algorithm 0.11.1: COMPUTEEUC2D$(city1, city2)$

$dx \leftarrow (city1 - city2)$

$dy \leftarrow (city1 - city2)$

return $((int)(sqrt((dx * dx) + (dy * dy)) + .05))$

0.12 Results from GRASP

Table 8: Results from GRASP

Size	Performance	Time (ms)	Known Optimum	% from Optimum
52	8011	149	7542	6.22
76	565	162	526	7.41
225	4125	1451	3916	5.34
280	1447	2320	2579	-43.89
299	52735	2598	48191	9.43
493	38090	7693	35002	8.82
657	54004	16001	48912	10.41
1291	56659	64729	50801	11.53
2103	93661	201217	80450	16.42
3795	29258	1286417	28772	1.69
5915	647829	3358824	565530	14.55
13509	22510022	22690851	19982859	12.65

GRASP was first introduced in the early to mid nineties as an optimization algorithm at AT&T labs. Looking at the table containing the GRASP data, it is easy to see that this algorithm is an excellent option to address this problem. Not only did it produce results extremely close to the known optimal on a majority of the problems, it required very little time and resources to do so. Given more time and some minor enhancements, it is likely that this very simple algorithm could come even closer to the known optimal values. Even on problems as large as 3795 cities, it was able to come within 1.69% of the known optimal solution, running only for approximately 29 seconds!

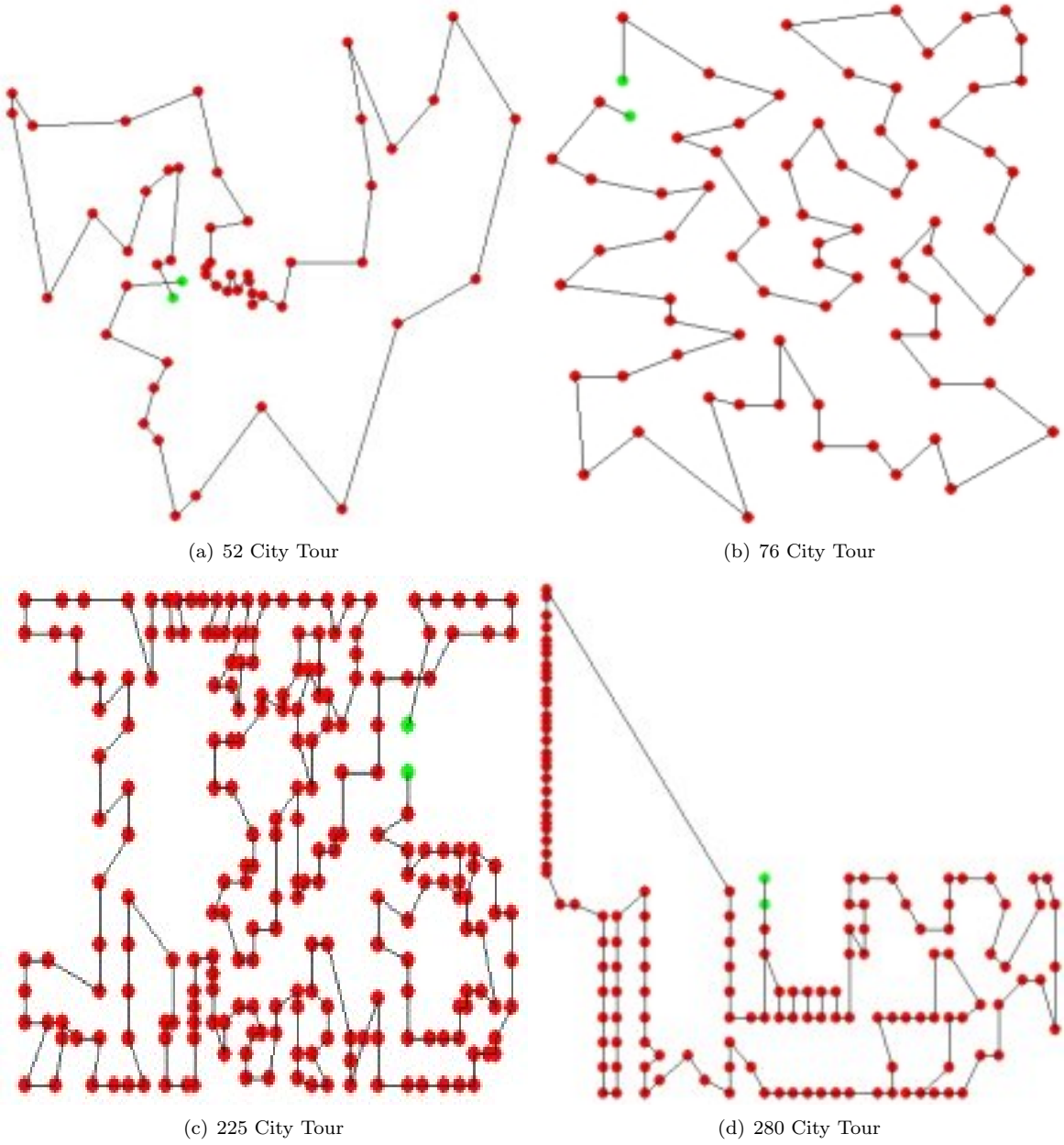
Looking at the smaller maps, it is easy to see that this algorithm was able to produce tours with little or no overlaps in the path the salesman is to take. If you compare GRASP's performance to that of SA, GRASP produced excellent results within only a few percentage points of SA at a significant fraction of the time and required memory.

0.13 Results from Greedy, SA, & RTS

This section details several tables. The first, Table 10 gives a side-by-side comparison of all the algorithms and their performance values but gives neither their run-time nor their percent from optimum values. The subsequent tables, 11 and 12, will give a closer look at specific performance metrics such as their running times and the percent from the known optimum tour lengths.

The annealing schedule used by the simulated annealing algorithm started with an initial temperature of 10 ($T = 10$) and was cooled at a rate of 1% every iteration, i.e., $T = T \cdot .99$. The step quantity used by RTS was dependent upon the problem size, refer to Table 9 for the step size. The red text in Table 9 are those maps that the RTS couldn't finish because of time constraints. The map with 2103 was run for a week with no results before the process was killed. It's also interesting to note that the larger maps were run with fewer steps and still couldn't finish in a timely matter, nor with decent results.

As is easily viewable from the greedy algorithm's table, the disadvantages of such a simple,



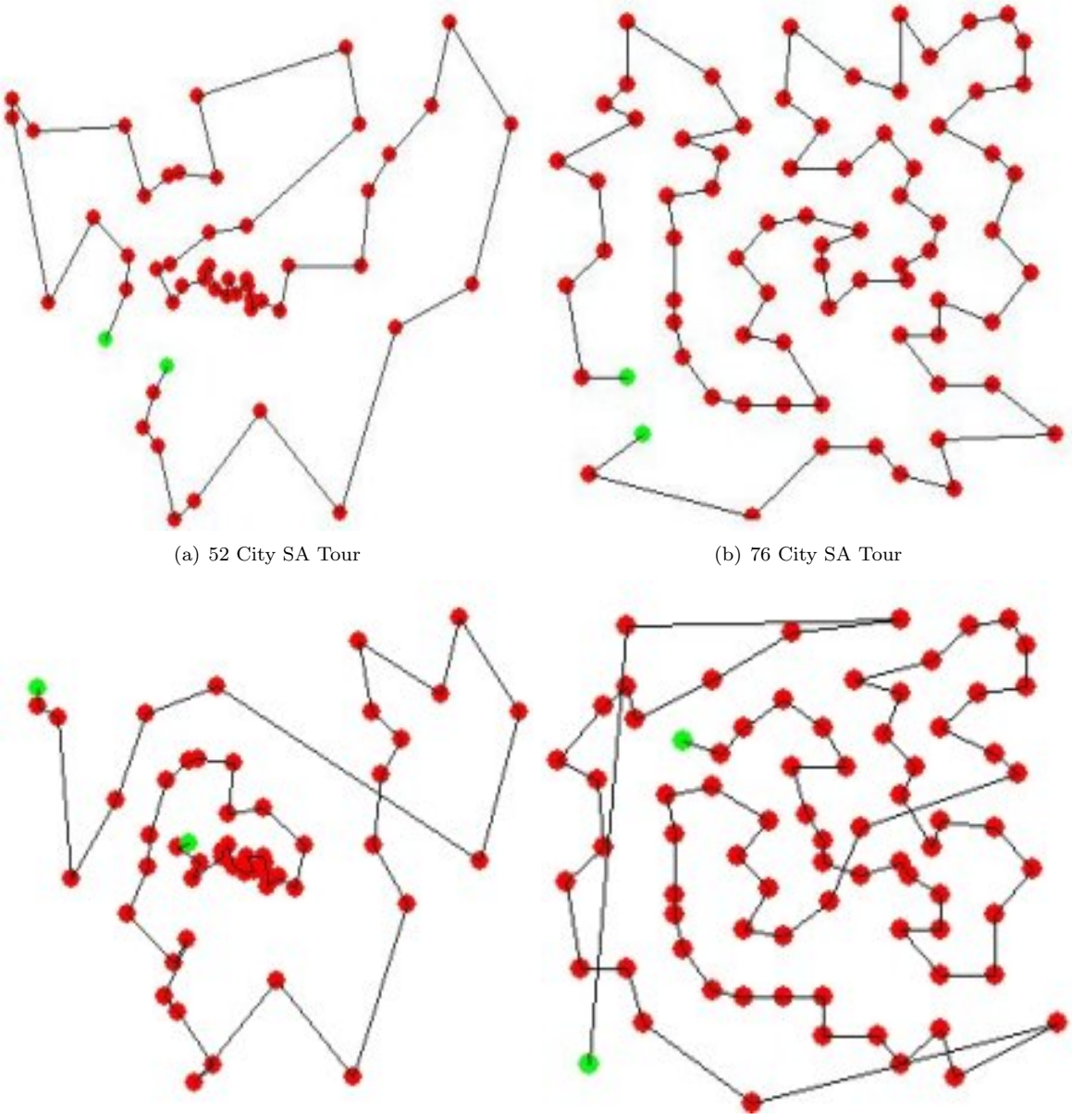
(a) 52 City Tour

(b) 76 City Tour

(c) 225 City Tour

(d) 280 City Tour

Figure 5: GRASP Created Tours, Map Sizes 52 - 280



(a) 52 City SA Tour

(b) 76 City SA Tour

(c) 52 City Greedy Tour

(d) 76 City Greedy Tour

Figure 6: SA & Greedy Created Tours, Map Sizes 52 - 280

Table 9: Step Size used by RTS Algorithm

Size	Step Count
52	100,000
76	150,000
225	400,000
280	400,000
299	400,000
493	400,000
657	800,000
1291	800,000
2103	500,000
3795	500,000
5915	500,000
13509	500,000

short-sighted local search are far outweighed by the advantages. This search produced decent tours using the least amount of resources. Looking at the tours produced, there is definitely room for improvement since every tour produced by this algorithm has multiple overlapping paths.

Simulated annealing is an excellent option on smaller problems but it isn't practical on larger ones due to time and performance constraints. SA definitely outperformed the greedy and RTS algorithms in terms of shortest tours, but required considerably more time as the problem size increased, making this search feasible only for the smaller sized problems. GRASP's performance was either better than or within several percentage points of SA, making GRASP

Table 10: Side-by-side comparison of Greedy, SA, GRASP & RTS

Size	Greedy	SA	GRASP	RTS	Best GA	Known Best
52	7795	8233	8011	15404	7766	7542
76	662	529	565	1274	540	526
225	4650	3895	4125	23810	4231	3916
280	1634	1381	1447	15657	1452	2579
299	59837	52748	53367	420242	55325	48191
493	43255	36437	38090	263060	43127	35002
657	60337	51191	54233	533125	61685	48912
1291	60078	55047	58773	1133795	747310	50801
2103	86383	85341	84666	-	2013056	80450
3795	32545	26271	31416	-	2595433	28772
5915	707156	625387	647829	-	34159617	565530
13509	24738755	22652227	22510022	-	1842132099	19982859

Table 11: Results from SA

Size	Performance	Time (ms)	Known Optimum	% from Optimum
52	8233	213	7542	9.16
76	529	981	526	0.57
225	3895	10607	3916	-0.54
280	1381	30588	2579	-46.45
299	52748	8924	48191	9.46
493	36437	38268	35002	4.1
657	50889	93196	48912	4.04
1291	54543	430840	50801	7.37
2103	84223	749122	80450	4.69
3795	26271	7003303	28772	-8.69
5915	625387	7837030	565530	10.58
13509	22652227	58416369	19982859	13.36

Table 12: Results from Greedy Algorithm

Size	Performance	Time (ms)	Known Optimum	% from Optimum
52	8963	1	7542	18.84
76	662	1	526	25.86
225	4650	1	3916	18.74
280	1634	1	2579	-36.64
299	59837	1	48191	24.17
493	43255	4	35002	23.58
657	60337	7	48912	23.36
1291	60078	28	50801	18.26
2103	86383	73	80450	7.37
3795	32545	674	28772	13.11
5915	707156	1638	565530	25.04
13509	24738755	8739	19982859	23.8

Table 13: Results from RTS Algorithm

Size	Performance	Time (ms)	Known Optimum	% from Optimum
52	15404	7288	7542	104.24
76	1274	20181	526	142.21
225	23810	366317	3916	508.02
280	15657	567837	2579	507.1
299	420242	583722	48191	772.03
493	263060	3311655	35002	651.56
657	533125	5372302	48912	989.97
1291	1133795	15841417	50801	2131.84
> 1291	Didn't run due to time constraints			

a much better alternative. On top of this, GRASP requires significantly less running time and memory resources.

Looking at Table 13 would easily lead one to conclude that this search isn't a feasible option, and I believe that to be a reasonable conclusion. On problems as small as 1291 cities, it required 1.5 million milliseconds just to complete. It was terminated after 1.5 weeks of working on the 2103 city problem without returning a solution. Despite all of the time it was given to run, RTS was definitely the worst performer of all the tested algorithms. In its pure form, RTS is a very poor choice for the TSP.

0.14 Results from Genetic Algorithm

Several new techniques were employed and tested by this thesis, namely adaptive restart, adaptive mutation, seeding, and the harem parent selection algorithms. In addition to these new techniques, varying the GA's population size and running time were also tested. The results and conclusions will be presented in this section.

0.14.1 Seeding vs. No Seeding

Table 14: Results of Seeding the GA's Population

Size	Crossover	% increase over non-seeded GA	% increase over over Greedy algorithm
52	OX	6	13
	PMX	35	5
76	OX	-1	17
	PMX	15	10
225	OX	13	13
	PMX∗	114	0
280	OX	47	13
	PMX∗	315	0
299	OX	30	9
	PMX∗	204	0
493	OX	85	1
	PMX∗	218	0
657	OX	215	.02
	PMX∗	432	0
> 657	No Improvement over Greedy Algorithm		

Note: The data from Table 14 were generated looking only at the genetic algorithm that was run for 10,000 generations using the harem-elitism parent selection with both the OX and PMX crossovers. The table doesn't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern. Looking at the table in Appendix A, the equation used to calculate the values in column 3 is: $\frac{NAMR-SNAMR}{SNAMR}$. The equation for column 4 is: $\frac{GREEDY-SNAMR}{SNAMR}$

Overall, the results of seeding the GA's population were very encouraging **when** the population was rather small (< 1291). As can be seen from the table, the GA was unable to improve the solution for any problem instance tested over 657 cities. Additionally, there is a trend that can be seen: there is an indirect relationship between the percent improvement of the seeded GA vs. the greedy algorithm. Specifically, as the city count grows, the overall performance, or improvement gained, drops steadily.

As previously stated, problem instances over 1,291 cities (≥ 1291), the GA was unable to improve upon the solution provided by the randomized greedy algorithm. Since the GA took significantly longer than the greedy algorithm did, this means there were a lot of wasted cycles. Overall, this finding is significant because it points to a problem with GAs and the TSP problem, at least when seeding the population. There seems to be a very clear point when using a GA stops yielding answers that are better than the simplest of neighborhood searches.

But there are several other conclusions that can be drawn here too. One of the most obvious is that the 2-point OX crossover is vastly superior to the PMX. On the table above, the asterisk ($*$) next to the crossover indicates that the crossover was incapable of improving the solution rendered by the greedy algorithm. As is clearly visible, the PMX crossover could not improve upon the solution provided by the greedy solution for problems over 76 cities. So why are there still such large improvements over the non-seeded GA? That is because the PMX crossover does so poorly that even a simple greedy search vastly outperformed the GA, sad considering the greedy algorithm needed only 7 ms to produce a solution on the 657 city

problem versus approximately 60,000 ms for the GA. The GA using the OX crossover was able to improve upon the greedy solution, but still needed in the neighborhood of 60,000 ms to finish.

0.14.1.1 Adaptive Mutation Rate

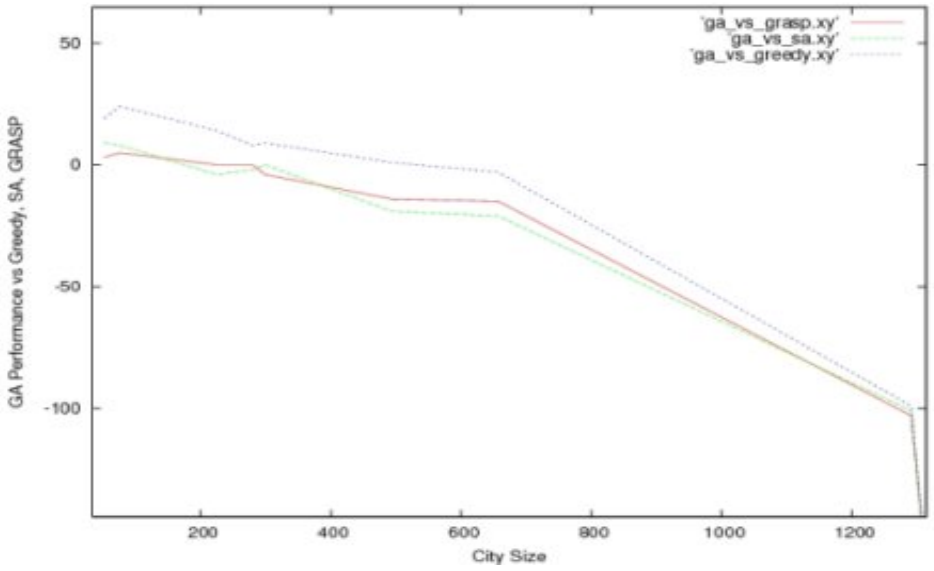
Table 15: Results of Adaptive Mutation Rate on Genetic Algorithms

Size	Crossover	% increase over NAMR
52	OX	1
	PMX	29
76	OX	-2
	PMX	-12
225	OX	-1
	PMX	5
280	OX	-5
	PMX	-5
299	OX	1
	PMX	2
493	OX	-3
	PMX	-5
657	OX	-4
	PMX	-2
1291	OX	1
	PMX	-1
2103	OX	.4
	PMX	.2
3795	OX	-.3
	PMX	-.1
5915	OX	-.02
	PMX	.2
13509	OX	-.02
	PMX	.2

Note: The data from Table 15 were generated looking only at the genetic algorithm that was run for 10,000 generations using the harem-elitism parent selection with both the OX and PMX crossovers. The table doesn't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern. Looking at the table in Appendix A, the equation used to calculate the values in column 3 is: $\frac{NAMR-AMR}{AMR}$.

The results of adaptive mutation rate were pretty conclusive - adaptive mutation rate

Figure 7: Left side shows the percent performance of the GA over/under the respective algorithm. Blue is GA vs. Greedy, Red is GA vs. GRASP, Green is GA vs. SA. As is visible from the graphs, the GA's performance significantly degrades over 1000 cities.



offered no benefit to the overall performance of the GA when applied to the TSP. A quick inspection of the results listed in the table and it's easy to see that there are no patterns, in fact, the values look fairly random. One might actually conclude that the adaptive mutation rate technique actually hurt the overall performance since the majority of table entries are negative. Despite the fact that the AMR technique required additional code checking (*if-then statements*), the difference in running time was very minimal/non-existent. This attempt to improve upon the GA's mutation operator was a partial failure. I use the word *partial* since the overall degradation in performance was very minimal, in most cases less than 1%, and in some cases it was actually positive.

Even though this technique failed on the TSP, is it possible that it could succeed on a different problem? Most likely not due to the fact that this technique is geared towards problems that can be encoded as permutations. Most problems that GAs are reported to work best on have bit encoding schemes, a technique that AMR does not translate well to.

0.14.1.2 Random Restart

Note: The data from Table 16 were generated looking only at the genetic algorithm that was run for 10,000 generations using the harem-elitism parent selection with both the OX and PMX crossovers. The table doesn't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern. Looking at the table in Appendix A, the equation used to calculate the values in column 3 is: $\frac{NAMR-AR(NAMR)}{AR(NAMR)}$.

While random restart is not a new concept, there are very few, if any, reports of this technique being applied to genetic algorithms (searching through the ACM database turned up no hits). Restart is an escape method, used to escape local optima when an algorithm prematurely converges, which is exactly how it was applied here. [1] used SOM's to mine the 'genealogy' for results. Their goal was the exact same, prevent their GA from prematurely converging. I just recreated a portion of the population to get a new influx of gene sequences

Table 16: Effects of Random Restart on Genetic Algorithms

Size	Crossover	% increase over non-restart.
52	OX	8
	PMX	24
76	OX	-3
	PMX	-15
225	OX	10
	PMX	25
280	OX	16
	PMX	25
299	OX	22
	PMX	1
493	OX	23
	PMX	1
657	OX	19
	PMX	3
1291	OX	.3
	PMX	1
2103	OX	.45
	PMX	< -1
3795	OX	< -1
	PMX	.1
5915	OX	5227
	PMX	5232
13509	OX	< 1
	PMX	< 1

that have most likely disappeared this late in the run, or ones that might not have been seen yet.

One look at the table and it's nearly impossible to miss the affect that random restart had on the performance of the GA. In general, restart produced excellent results; for those graphs that restart didn't do too well on, it didn't degrade performance noticeably either. The maps that did not do as well as the average can likely be explained away as maps with tough features to overcome and given more time and iterations would likely have reported better results.

0.14.1.3 Varied Population Sizes

Table 17: Effects of Different Population Sizes on Genetic Algorithms

Size	Crossover	% increase of 50000 over 10000
52	OX	7
	PMX	19
76	OX	-5
	PMX	-11
225	OX	7
	PMX	2
280	OX	35
	PMX	4
299	OX	26
	PMX	-8
493	OX	33
	PMX	-5
657	OX	48
	PMX	2
1291	OX	29
	PMX	7
2103	OX	10
	PMX	7
3795	OX	5
	PMX	5
5915	OX	54
	PMX	54
13509	OX	2
	PMX	2

Note: The data from Table 17 were generated looking only at the genetic algorithm that was run for 10,000 generations using the harem-elitism parent selection with both the OX

and PMX crossovers. The table doesn't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern. Looking at the table in Appendix A, the equation used to calculate the values in column 3 is: $\frac{NAMR-LP_NAMR}{LP_NAMR}$.

The GA needs a population of at least $(N-1)/2$ members in order to sample all possible edges at least once [26]. So it makes sense that larger populations should outperform smaller ones since there is a greater chance of encountering all of the edges at least once, and in general, table 17 supports this conclusion. Why the anomaly for the map with 76 cities? I can only speculate that the combination of crossover and mutation operators just wasn't enough to overcome the complexities in solving this particular map.

This boost in performance doesn't come cheap though, the average running time nearly quadrupled: 50,000 ms for one instance vs. 190,000 ms for the equivalent instance with 500 members. So we have the running time quadrupling (more or less), the population size increased by 333%, but the performance gains only reached double digit levels even in the best case. But, it worked in increasing the GA's overall performance, it just wasn't cheap.

0.14.1.4 Harem Selection vs. The Others

Note: The data from Table 18 were generated looking only at the genetic algorithm that was run for 10,000 generations using the harem-elitism parent selection with both the OX and PMX crossovers. The table doesn't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern.

The results from using harem select are very encouraging, it outperformed every other selection technique on all but the largest of the problems. When compared to the three lowest performing selection algorithms, which also happen to be the most commonly reported on and used, there was a minimum of a 200% performance gap for any city size less than 500 cities! An interesting pattern to take note of was the shrinking of the performance gap between all of the

Table 18: Percent Difference When Compared to Harem-Elitism

Size	Crossover	Elitism	HaremR	Roulette	Stochastic	Tournament
52	OX	-6	-2	179	193	169
	PMX	-15	-22	72	102	96
76	OX	4	2	244	273	192
	PMX	6	10	155	157	172
225	OX	4	12	680	692	628
	PMX	5	51	241	266	243
280	OX	5	5	1181	1213	1137
	PMX	4	50	273	307	288
299	OX	6	9	771	767	734
	PMX	13	72	252	277	256
493	OX	2	20	422	423	412
	PMX	17	61	199	202	196
657	OX	1	45	312	314	299
	PMX	14	58	141	145	135
1291	OX	2	11	72	72	68
	PMX	4	29	69	69	67
2103	OX	< 1	5	40	39	38
	PMX	1	15	40	40	40
3795	OX	< 1	3	27	27	27
	PMX	< 1	8	27	27	26
5915	OX	< -1	2	13	12	11
	PMX	< -1	2	12	13	12
13509	OX	< -1	2	13	12	11
	PMX	< -1	2	12	13	12

selection techniques as the map sizes grew. I believe this can easily be attributed to the fact that the GA just didn't fare well on any large size problems, irregardless of the techniques used. In fact, if you look at the values in Appendix A and compare those to those returned by even a simple greedy search, and the results returned by the GA look almost random. Remember, in order for a GA to perform well on a problem of those sizes, it must contain a population large enough to contain all of the possible $N(N-1)/2$ edges!

Overall, the rankings seem to put the harem-elitism selection algorithm at the top followed by pure elitism, harem-roulette. Roulette, stochastic uniform, and tournament all roughly tied for fourth.

The fact that harem-roulette, a hybrid elitist-stochastic selection algorithm, significantly out-performed roulette and the other purely stochastic selection techniques (stochastic uniform and tournament) would indicate that some form of elitism will yield better overall results than a purely stochastic search will.

Another interesting thing to note is tournament select's speed. Even on the 13,509 city problem, tournament select required only 16,000 ms vs. an average of 1,150,000 ms for the other 5! It might be tempting to think that simply extending its running time will improve performance, this was not the case - doubling its running time yielded less than a 1% improvement in solution quality, hardly worth the effort!

0.14.1.5 Crossovers: EAX, MOX, PMX, & OX

Note: The data depicted in Table 19 was generated looking only at the genetic algorithm that was run for 10,000 generations. The graphs don't include results from other techniques such as adaptive mutation rate, restart, or the hybrid GA with local search, though their results were very similar and followed the exact same pattern.

On the larger problems, edge assembly crossover (EAX) did extremely well, often outperforming the other crossover algorithms by as much as 32% when compared to the next closest. Looking at the unaltered distance values in Appendix A, you will see that in many of the maps

Table 19: Percent Difference from Known Optimal (lower is better) for Differing Crossovers

Size	Crossover	HaremE	HaremR	Elitism
52	OX	12	10	5
	PMX	53	19	53
	EAX	29	6	12
	MOX	33	23	40
76	OX	6	9	11
	PMX	32	45	37
	EAX	23	21	9
	MOX	55	53	45
225	OX	28	48	34
	PMX	216	397	233
	EAX	53	20	28
	MOX	165	415	199
280	OX	-17	-13	-13
	PMX	166	298	176
	EAX	-32	-32	-32
	MOX	124	282	145
299	OX	61	76	70
	PMX	278	551	329
	EAX	56	37	41
	MOX	256	532	314
493	OX	126	171	130
	PMX	295	535	363
	EAX	124	94	108
	MOX	287	545	536
657	OX	300	298	477
	PMX	665	573	962
	EAX	277	274	401
	MOX	636	537	949

that EAX was run against, the distance values it returned were comparable to those returned by using random restart or the hybrid GA/Local search (memetic algorithm) versions of the GA!

While EAX performed extremely well, it requires copious amounts of time in order to produce these results. If you look at how the algorithm works as detailed in the section on genetic algorithms, you will understand why it requires the amounts of time it does. The algorithm has to build a table for every parent combination chosen and then use its distance heuristics when deciding which edge to chose. Running the GA using the roulette parent selection technique with either PMX or OX took approximately 121,000 ms vs. 115,000,000 ms for EAX or 3,000,000 ms for MOX. So for larger problem sizes, neither EAX nor MOX are appropriate choices if time is a factor.

0.14.1.6 Combination Results - GA with Local Search

Excluding the adaptive mutation rate (AMR) technique, the other new techniques produced excellent results when compared to a plain, run of the mill genetic algorithm. This section sought to test if we could further the performance even more so by combining all the techniques at once. That is, run the GA using random restart coupled with the local search (replacing the mutation operator with the 2-Opt search algorithm), all in conjunction with the new parent selection technique, harem elitism/roulette. The run time was held constant, as before, at 10,000 generations with a population size of 150. The following table gives the results of this run.

Note: The data depicted in Table 20 was generated looking only at the genetic algorithm that was run for 10,000 generations. OX was the only crossover used, and the GA used both adaptive restart and local search. Looking at the table in Appendix A, the equation used to calculate the values in column 3 is: $\frac{(COMBO - KNOWN_BEST)}{KNOW_BEST} * 100$

0.14.1.7 Genetic Algorithms and the TSP

When looking at Table 21, the values used for the GA column are calculated using only the time values for the OX forms of elitism and the two harem selects. When the GA used the

Table 20: Percent Difference from Known Optimal (lower is better) when using GA/Local Search

Size	Crossover	HaremE	HaremR	Elitism
52	NAMR	5	4	7
	SNAMR	6	5	5
76	NAMR	5	5	5
	SNAMR	6	6	5
225	NAMR	10	8	9
	SNAMR	3	3	2
280	NAMR	-42	-43	-42
	SNAMR	-45	-45	-43
299	NAMR	24	26	20
	SNAMR	12	9	8
493	NAMR	39	34	32
	SNAMR	14	9	12
657	NAMR	66	55	52
	SNAMR	13	14	12
1291	NAMR	186	151	128
	SNAMR	17	16	16
2103	NAMR	274	201	195
	SNAMR	5	6	6
3795	NAMR	336	248	215
	SNAMR	12	12	10
5915	NAMR	373	360	277
	SNAMR	24	25	23
13509	NAMR	410	373	306
	SNAMR	24	24	23

Table 21: Average Running Times (in ms) for the Various Test Algorithms

Size	Greedy	Grasp	GA	SA	RTS
52	< 1	197	3518	213	7288
76	< 1	190	6323	981	20181
225	1	1670	14322	10607	366317
280	1	2573	18129	30588	567837
299	1	3006	19401	32924	583722
493	4	8871	35402	38268	3311655
657	7	17196	59143	93196	5372302
1291	28	78152	107722	430840	15841417
2103	73	230862	179820	749122	-
3795	674	1526608	339052	7003303	-
5915	1638	3830209	529836	7837030	-
13509	8739	25428285	1243551	58416369	-

PMX crossover algorithm, the time to complete was approximately 10% longer. On average, the tournament select algorithm was 100 times faster than the values listed in the table. So for the 13,509 city problem, the tournament select algorithm only took 16,059 milliseconds to complete vs. the 1,243,551 ms average for the other forms. The stochastic remainder parent select algorithm was on average 20% faster than those listed (elitism & harem select, both forms). Even when choosing amongst the various GA configurations there is a trade-off. If speed is a concern, then the tournament selection algorithm is the best bet. The trade-off here is with performance. If performance is your main concern, then any one of top three parent selection algorithms, elitism, harem-roulette, or harem-eletism, will work just fine.

When comparing the GA's performance to that of the other metaheuristic algorithms, a genetic algorithm might not be your first choice for problems such as the TSP as is demonstrated by Figure 7. It was able to beat the other algorithms, i.e., greedy, GRASP, SA, and RTS, in its pure form (no local search or restart) for the problem instances of 225 cities or less. Starting at 225 and going larger, the GA's performance started to falter. For problem sizes from 225 to 657, the various tables show the GA's performance was very comparable to the others, especially when you consider the EAX crossover the 2-Opt/GA hybrid.

This brings up a couple of very important questions - *When should I use genetic algorithms?* and *Will new advances in genetic algorithms expand their usefulness beyond the answer to the first question?* Before addressing these questions, let's reconsider something that was stated earlier - the TSP was merely used as an "engine of discovery", so our main goal was not necessarily to solve this problem to optimality, but to prove my improvements to the genetic algorithm could improve the overall performance of the algorithm.

I believe the main issue with genetic algorithms is that there are few jobs a GA can do better than a heuristic-based search, as this thesis has demonstrated. Generally speaking, if you have an idea of how to solve a problem, you're better off implementing that idea than you are turning a GA loose on your problem. Genetic algorithms are random by design; this stochastic nature doesn't generally lend itself well to a clean-cut problem, intractable as that problem may

be. But there are issues that may warrant their use: noise and data access.

By noise, I mean that the evaluation of a given candidate solution may vary randomly. Great examples of noisy data would include game play where a high score doesn't mean you achieved your goal. Other examples include finding patterns in financial data or assembly line automation, etc. GAs deal with noise well because they are slower to react to seemingly fantastic or abysmal evaluations. Additionally, the noise will tend to get smoothed out over many candidates, populations, and generations.

Data access means that you may not have the access and/or ability to get at the information you want into order to make good decisions with other more obvious types of AI. Returning to the game play example, when humans play StarCraft they usually set up some sort of defense for their base depending on what type of attack they're expecting. Any heuristic algorithm needs to have an analysis of the defense in order to produce a recommended offense, and writing that programmatic analysis could be extremely difficult.

Genetic algorithms do have the advantage that with an appropriate encoding they can invent solutions that don't require the intervening abstraction. Thus, if you can't discover a way of abstracting out the problem, you can get a GA to bypass it. To accomplish this usually means making the encoding flexible enough (read: has a large enough search space) to encompass all decent solutions and then hoping it finds one.

They also have the benefit that they can find non-intuitive solutions but they are computationally expensive, and work better the more resources you can throw at them. Larger populations and more generations will give you better solutions. This means that GAs are better used offline. Returning again to the game play example, one way of doing this is by doing all of the GA work in-house and then releasing an AI tuned by the GA. Another idea is to have the GA work a lot like your screen saver - it could run on the user's computer while the computer is not being used. This way, the game can tune the game play to each specific user in the most efficient way.

0.14.2 Benefits of GAs

There are many ways to speed up and improve a GA-based application as knowledge about the problem domain is gained. Also, it's easy to exploit previous or alternate solutions as was demonstrated by seeding the population and having the GA attempt to improve upon that. While this only worked on smaller instances of the problem set, [18] did have good results on the rectilinear problem. So it's only natural to assume this technique should transfer nicely to other problems. A genetic algorithm is composed of flexible building blocks that allow for hybrid applications, e.g. GA/2-Opt hybrid algorithm! On top of all of this, GAs have a substantial history and range of use.

0.14.3 When to use Genetic Algorithms?

Genetic Algorithms can be applied to a variety of problems. Problems which either can not be formulated in exact and accurate mathematical forms or may contain noisy data are excellent candidates for a GA. Problems that take too much computational time or are impossible to solve for traditional computational systems will also be great candidates for a GA. A class of problems that fit these descriptions well are multi-objective optimization problems. These problems have several optimization goals that must be achieved simultaneously. A commonly studied example in academia would be TSP with timeslots. Here there are two goals - minimize the distance traveled but also ensure the salesman hits certain cities within certain time windows.

0.14.4 Disadvantages of GAs

GAs are nondeterministic in nature and as such, it is not guaranteed that a GA will return the same solution or an optimal solution in each run. GAs have a weak theoretical basis and require proper turning for their many parameters in order to achieve good performance. Even once you have found a good tuning, GAs are computationally expensive and thus are (very) slow.

0.15 Conclusion

I have presented and investigated several new methods of improving a genetic algorithm while comparing its overall results to several other metaheuristic algorithms in context of the TSP. Techniques investigated included a new parent selection algorithm, harem select in conjunction with population seeding, a new mutation operator, a hybrid GA, and random restart among others. In addition to this, a newer algorithm, GRASP, was presented as a viable alternative to TSP.

My test results show the new harem select produced significant improvements over the standard selection algorithms commonly employed. It improved the distance metric anywhere from 11 to 1100% compared to roulette, stochastic uniform, or tournament selection and 1 to 15% when compared to pure elitism. I also confirmed Justrom's work on population seeding [18]. Seeding the population improved the results up to 615% depending on the problem size.

0.16 Future Directions

0.16.1 Evaluation Strategies

A more complicated evaluation strategy might look at the number of overlaps in the current path and add a small penalty for each one in addition to the total distance traveled. This evaluation strategy would award paths that don't backtrack and crossover previously traveled paths in addition to being shorter.

0.16.2 Termination Criteria

The number of generations was this GA's only termination criteria. This is a rather simplistic method when compared to others available such as meeting a performance goal (i.e., finding a solution within a stated percent, e.g. 1%, of known optimal) or ending once the GA stopped improving the solution from generation to generation.

There are positives and negatives to the method just mentioned. Using a fixed number

of generations is very simple and straight forward to implement. Additionaly, it guarantees the GA will end without any tricky logic, but it can result in the GA running longer than absolutely necessary. A relevant example would be the GA stopped improving the solution after 2,000 generations but was forced to run for 10,000. Also, ending the GA after so many generations of non-improvement ignores the random nature of a GA and its ability to climb out of local optima, but can be the most efficient in that it ends the GA at the first sign the GA has converged on a local optima.

Forcing the GA to achieve a certain goal before termination doesn't guarantee the GA will ever terminate and requires a secondary check to determine when to give up. Additionally, this method isn't usable for problems where the lower bound is unknown or can't be calculated.

0.16.3 Dominance and Diploidy

I could carry the genetic analogy even further and use such techniques as diploid (double-stranded) chromosomes and dominance. Dominance would involve an operator that would look at the alleles in a given gene position where one gene is dominant and all others are recessive. This could be done with the diploid method or through the phenotype. My GA is haploid, single-stranded, and has a higher tendency to destroy "recessive" genes since there is only one position for any given gene. The diploid route would ensure a higher survival rate for recessive genes which could translate to better performance on problems with rapidly changing fitness maps.

0.16.4 Parallelization

Using parallel computers or taking advantage of the newer multi-core chips could greatly improve the run-time performance of GAs. Most GA codes involve very little parallelization. Those that do generally employ a master/worker hierarchy where the master performs the parent selection and mating and passes the task of fitness evaluation off to the workers. This technique relies on access to multiple computer and communication via TCP/IP. With the mass

introduction of multi-core chips, GAs would greatly benefit from "balancing" algorithms that distribute the fitness evaluations among the available processors within a single system.

0.16.5 Multimodal Optimization

Problems like TSP have a solution space that is multimodal, i.e., has several peaks. A GA will tend to converge to one of those peaks, particularly if one is more fit than the others. A technique to explore would be to have the GA converge on several of the peaks simultaneously. Instead of devoting the entire population to finding one solution, it is used instead to find several.

As this happens, one could logically think of different "species" developing from the original population. Groups of members, or subpopulations, will become better adapted for the various peaks that are converged upon. Likewise, each peak could be thought of as a seperate environment/island, etc. With this type of specialization/speciation occuring, it would be reasonable to simply end the GA with multiple solutions or to wait for the "species" to be better developed and start cross-breeding the species.

Bibliography

[1] Heni Ben Amor and Achim Rettinger. Intelligent exploration for genetic algorithms, using self-ogranizing maps in evolutionary computation. In GECCO 2005, pages 1531–1538, 2005.

[2] Peter Anderson and Daniel Ashlock. Advances in ordered greed. In ANNIE Conference, St. Louis, MO, USA, November 2004.

[3] R. Battiti and G. Tecchiolli. The reactive tabu search. ORSA Journal on Computing, 6(2):126–140, 1994.

[4] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Survey, 35(3):268–308, Sep 2003.

[5] E. K. Burke and A. J. Smith. A memetic algorithm to schedule planned maintenance for the national grid. J. Exp. Algorithmics, 4:1, 1999.

[6] Applegate D, Bixby R, Chvatal V, and Cook W. Georgia tech web site that hosts the concorde project. http://www.tsp.gatech.edu//methods/opt/zone.htm.

[7] Applegate D, Bixby R, Chvatal V, and Cook W. Implementing the tantzig-fulkerson-johnson algorithm for large traveling salesman problems. Mathematical Programming, 97:91–153, 2003.

[8] G. B. Dantzig, R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling salesman problem. Operations Research, 2:393–410, 1954.

[9] Larry J. Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In FOGA, pages 265–283, 1990.

[10] Xavier Gandibleux, Marc Sevaux, Kenneth Sorensen, and Vincent T'kindt. Metaheuristic for Multiobjective Optimisation. SPRINGER, 2003.

[11] Fred Glover. Future paths for integer programming and linds to artificial intelligence. Computers and Operations Research, 13:533–549, 1986.

[12] D.E. Goldberg. Genetic Algorithms in Search Optimisation and Machine Learning. Addison Wesley, 1989.

[13] J. Gottlieb and T Krusei. Selection in evolutionary algorithms for the traveling salesman problem. In SAC 2000, pages 415–421, March 2000.

[14] John Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic algorithms for the traveling salesman problem. In John Grefenstette, editor, Proceedings of the First International Conference on Genetic Algorithms and Their Applications, 1985.

[15] M. Grotschel. On the symmetric travelling salesman problem: solution of a 120-city problem. Mathematical Programming, 12:61–77, 1980.

[16] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees: Part ii. Mathematical Programming, 1:6–25, 1971.

[17] Chen hsiung Chan, Sheng-An Lee, Cheng-Yan Kao, and Huai-Kuang Tsai. Improving eax with restricted 2-opt. In GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1471–1476, New York, NY, USA, 2005. ACM Press.

[18] Bryant A. Julstrom. Seeding the population: Improved performance in a genetic algorithm for the rectilinear steiner problem. Proceedings of the 1994 ACM Symposium on Applied Computing, pages 222–226, 1994.

[19] Byrant Julstrom. Very greedy crossover in a genetic algorithm for the traveling salesman problem. In Proceedings of the 1995 ACM symposium on Applied computing, pages 324–328, Feb 1995.

[20] Tsplib - web based repository containing the various maps used in this thesis. http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.

[21] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. Science, 220(4598):671–680, May 1983.

[22] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. Operations Research, 21:498–516, 1973.

[23] Web-based repository on links and research papers for memetic algorithms. http://www.densis.fee.unicamp.br/ moscato/memetic_home.html.

[24] Keith E. Mathias and L. Darrell Whitley. Genetic operators, the fitness landscape and the traveling salesman problem. In PPSN, pages 221–230, 1992.

[25] H. Muhlenbein. Evolution in time and space – the parallel genetic algorithm, 1991. citeseer.ist.psu.edu/muhlenbein91evolution.html.

[26] Yuichi Nagata and Shigenobu Kobayashi. Edge assembly crossover: A high-power genetic algorithm for the traveling salesman problem. In Proceedings of the 7th International Conference on Genetic Algorithms, pages 450–457, July 1997.

[27] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Experiments with simulated annealing. In Proceedings of the 22cd ACM/IEEE conference on Design automation, pages 748–752, June 1985. Las Vegas Nevada.

[28] Surendra Nahar, Sartaj Sahni, and Eugene Shragowitz. Simulated annealing and combinatorial optimization. In Proceedings of the 23rd ACM/IEEE conference on Design automation, pages 293–299, 1986.

[29] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM, 33:60–100, 1991.

[30] Thomas Stutzle. Local search algorithms for combinatorial problems - analysis, algorithms, and new applications. Master's thesis, Am Fachbereich Informatik der Technischen Universitat Darmstadt, December 1998.

[31] Jean-Paul Watson, C. Ross, V. Eisele, and J. Denton. The traveling salesrep problem, edge assembly crossover, and 2-opt. In PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature, pages 823–834, London, UK, 1998. Springer-Verlag.

[32] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. Transactions of the Institute of Electronics, Information and Communication Engineers, J83-D-1(1):3–25, 2001.

AppendixA

Notes: MOX and EAX were not tested on problem sizes over 1291 cities due to time constraints. Running the GA using the roulette parent selection technique with either PMX or OX took approximately 121,000 ms vs. 115,000,000 ms for EAX or 3,000,000 ms for MOX. The known best distances were calculated using the C programming language and were taken from the TSPLIB [20] website. As is visible for the 76 city problem, using Java's *long* primitive, the total distance for this map was computed at 526 versus the 538 listed on the site. This discrepancy I'm assuming is due to rounding errors between the two languages. A lot of the other maps did not have the shortest known path listed, so the lower bounds could not be verified. Hence, the bounds listed are most likely only approximately the same as those shown (look at the 280 city map, there are several values listed that are lower than the lowest bound. I do have the path used to get that bound, so it is non-reproducible).

Table 22: Results Table for GA

AMR: Adaptive Mutation Rate, SAMR: Seeded Adaptive Mutation Rate
NAMR: No Adaptive Mutation Rate, SNAMR: Seeded No Adaptive Mutation Rate
AR: Adaptive Restart, GALS: GA with Local Search
LP: Large Population, NI: No Improvement

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	PMX									
	elitism	11402	9842	8594	8505	9871	11736	10470	19231	7542
	harem_e	8958	11514	8594	8499	10592	9623	9315	21469	7542
	harem_r	9997	8960	8790	NI	14166	15202	9816	12436	7542
	roullete	18130	19770	NI	NI	17653	18051	18383	20033	7542

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	stochastic	24649	23214	NI	NI	21973	22192	23040	15707	7542
	tournament	22105	22613	NI	NI	22979	23270	12100	9930	7542
	OX									
	elitism	7958	7935	7730	8088	8282	8021	8545	7949	7542
	harem_e	8372	8417	8136	7965	7845	7821	7766	7779	7542
	harem_r	8369	8272	7957	8094	8512	8100	8425	7526	7542
	roullete	22350	23449	NI	NI	23317	22839	21962	8921	7542
	stochastic	23334	24666	NI	NI	23963	23583	24106	10008	7542
	tournament	20421	22677	8800	8854	23584	23097	12352	7867	7542
	EAX									
	elitism	9523	8477	-	-	-	-	-	-	7542
	harem_e	8302	9758	-	-	-	-	-	-	7542
	harem_r	8415	8001	-	-	-	-	-	-	7542
	MOX									
	elitism	11129	10589	-	-	-	-	-	-	7542
	harem_e	10409	10017	-	-	-	-	-	-	7542
	harem_r	9508	9277	-	-	-	-	-	-	7542
76	PMX									
	elitism	719	739	576	592	731	723	750	1513	538 (526)
	harem_e	793	695	602	603	833	758	818	1770	538 (526)
	harem_r	686	761	NI	NI	1256	1235	724	1069	538 (526)
	roullete	1780	1779	NI	NI	1573	1579	1796	1949	538 (526)
	stochastic	2021	1987	NI	NI	2002	1965	2061	1315	538 (526)
	tournament	1797	1839	NI	NI	1890	2037	1219	846	538 (526)
	OX									
	elitism	560	585	544	558	593	560	604	555	538 (526)
	harem_e	570	560	570	566	574	570	575	562	538 (526)
	harem_r	570	572	571	552	590	583	571	540	538 (526)
	roullete	2008	1930	NI	NI	1969	1910	1997	683	538 (526)
	stochastic	1982	2093	NI	NI	2038	2092	2128	842	538 (526)

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	tournament	1538	1638	581	669	1697	1789	1202	538	538 (526)
	EAX									
	elitism	614	639	-	-	-	-	-	-	526
	harem_e	641	646	-	-	-	-	-	-	526
	harem_r	560	573	-	-	-	-	-	-	526
	MOX									
	elitism	692	805	-	-	-	-	-	-	526
	harem_e	777	814	-	-	-	-	-	-	526
	harem_r	770	764	-	-	-	-	-	-	526
225	PMX									
	elitism	11096	10704	NI	NI	12251	11884	9787	28101	3916
	harem_e	9705	10229	4766	NI	10192	9941	8601	25940	3916
	harem_r	15854	15479	NI	NI	19409	19090	15877	19929	3916
	roullete	35112	34858	NI	NI	33809	34629	35093	31788	3916
	stochastic	37126	36846	NI	NI	36669	36137	36439	23850	3916
	tournament	35103	35120	NI	NI	36621	36160	28814	14116	3916
	OX									
	elitism	5039	4899	4165	4128	4330	4374	4588	4293	3916
	harem_e	4788	4721	4116	4129	4546	4231	4451	4236	3916
	harem_r	5069	5300	4271	4184	4445	4315	4454	4110	3916
	roullete	36680	36846	NI	NI	35596	35935	36386	6421	3916
	stochastic	36636	37426	NI	NI	36424	37334	36484	10627	3916
	tournament	34722	34377	NI	NI	35404	36363	28745	4307	3916
	EAX									
	elitism	5133	4727	-	-	-	-	-	-	7542
	harem_e	5425	5451	-	-	-	-	-	-	7542
	harem_r	4512	4496	-	-	-	-	-	-	7542
	MOX									
	elitism	9803	9728	-	-	-	-	-	-	7542
	harem_e	8435	8727	-	-	-	-	-	-	7542

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	harem_r	15833	16019	-	-	-	-	-	-	7542
280	PMX									
	elitism	6988	7108	NI	NI	7858	7142	6835	14354	2579
	harem_e	7221	6853	NI	NI	7010	7048	5478	22927	2579
	harem_r	10635	10261	NI	NI	12034	12118	9965	14482	2579
	roullete	26007	25565	NI	NI	25167	24029	26054	17423	2579
	stochastic	27965	27874	NI	NI	27472	27495	28103	13588	2579
	tournament	26491	26565	NI	NI	26869	25562	20601	8718	2579
	OX									
	elitism	2533	2234	1627	1590	1656	1579	1896	1679	2579
	harem_e	2247	2131	1486	1452	1543	1569	1837	1586	2579
	harem_r	2320	2247	1463	1508	1631	1566	1854	1513	2579
	roullete	27716	27301	NI	NI	26548	27076	27380	2850	2579
	stochastic	28380	27971	NI	NI	26813	27117	27881	6524	2579
	tournament	25537	26370	NI	1641	27423	26526	19122	1772	2579
	EAX									
	elitism	1822	1748	-	-	-	-	-	-	2579
	harem_e	1648	1760	-	-	-	-	-	-	2579
	harem_r	1739	1748	-	-	-	-	-	-	2579
	MOX									
	elitism	7159	6318	-	-	-	-	-	-	2579
	harem_e	5242	5779	-	-	-	-	-	-	2579
	harem_r	9867	9844	-	-	-	-	-	-	2579
	PMX									
	elitism	196013	206499	NI	NI	240637	225823	207439	407984	48191
	harem_e	186355	182063	NI	NI	203097	198641	183581	435955	48191
	harem_r	295022	313819	NI	NI	350863	338257	306709	334454	48191
	roullete	664713	641546	NI	NI	618726	635908	637213	511943	48191
	stochastic	689537	687270	NI	NI	670441	660893	669116	357377	48191
	tournament	645731	648912	NI	NI	675473	665570	535626	245255	48191

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	OX									
	elitism	79658	81997	56828	56778	65110	64516	65521	61423	48191
	harem_e	77058	77739	55325	54663	59494	61586	63707	57921	48191
	harem_r	87606	84696	56960	57063	65477	63504	67822	55813	48191
	roullete	685812	677752	NI	NI	677044	686135	666634	94829	48191
	stochastic	645731	674562	NI	NI	687040	672262	669197	184159	48191
	tournament	649974	648737	NI	59752	680295	659885	529437	60326	48191
	EAX									
	elitism	63562	68008	-	-	-	-	-	-	48191
	harem_e	74522	75071	-	-	-	-	-	-	48191
	harem_r	70121	64389	-	-	-	-	-	-	48191
	MOX									
	elitism	201515	199461	-	-	-	-	-	-	48191
	harem_e	172474	171324	-	-	-	-	-	-	48191
	harem_r	299679	304757	-	-	-	-	-	-	48191
493	PMX									
	elitism	159115	162154	NI	NI	163926	168657	154873	292612	35002
	harem_e	145805	138439	NI	NI	147201	146142	137736	291075	35002
	harem_r	232550	222446	NI	NI	230494	232352	223195	217240	35002
	roullete	403400	413573	NI	NI	406972	403938	405958	341718	35002
	stochastic	418027	418248	NI	NI	418805	410349	415449	247381	35002
	tournament	404089	410462	NI	NI	413058	411693	358281	165015	35002
	OX									
	elitism	79924	80529	43127	43194	59580	60004	63904	47189	35002
	harem_e	81740	79205	43170	42870	63202	59466	64450	47401	35002
	harem_r	99781	95080	43228	43169	68927	68139	77411	44063	35002
	roullete	412479	414162	NI	NI	416176	415282	407013	64782	35002
	stochastic	414563	414657	NI	NI	413476	411656	406560	95846	35002
	tournament	404324	405442	NI	NI	406938	409606	357988	46234	35002
	EAX									

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	elitism	71280	72758	-	-	-	-	-	-	35002
	harem_e	76431		-	-	-	-	-	-	35002
	harem_r	73401	67935	-	-	-	-	-	-	35002
	MOX									
	elitism	159290	159651	-	-	-	-	-	-	35002
	harem_e	136381		-	-	-	-	-	-	35002
	harem_r	225364	225772	-	-	-	-	-	-	35002
657	PMX									
	elitism	369057	374193	NI	NI	364661	363471	359195	578014	48912
	harem_e	335009	329411	NI	NI	328084	322215	320356	782194	48912
	harem_r	522448	519604	NI	NI	500450	494126	509453	347200	48912
	roullete	801735	794019	NI	NI	793142	778687	799470	698915	48912
	stochastic	803171	808259	NI	NI	798269	796140	792810	517378	48912
	tournament	785653	773914	NI	NI	804199	805332	728691	315801	48912
	OX									
	elitism	199333	195683	61865	61721	136465	125757	172549	72988	48912
	harem_e	202723	194715	61685	61854	131891	131162	163029	82463	48912
	harem_r	279210	282171	NI	61796	163286	155024	257510	74293	48912
	roullete	802710	801730	NI	NI	797172	796046	802459	103779	48912
	stochastic	802382	803587	NI	NI	800836	797195	795609	138662	48912
	tournament	783284	777114	NI	NI	792234	798958	714815	77582	48912
	EAX									
	elitism	176913	-	-	-	-	-	-	-	48912
	harem_e	178292	-	-	-	-	-	-	-	48912
	harem_r	251381	-	-	-	-	-	-	-	48912
	MOX									
	elitism	358475	-	-	-	-	-	-	-	48912
	harem_e	314936	-	-	-	-	-	-	-	48912
	harem_r	519894	-	-	-	-	-	-	-	48912
	PMX									

Continued on NextPage

Continued from Previous Page

Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	elitism	1014005	1012140	NI	NI	958480	956970	993769	1185069	50801
	harem_e	985389	977899	NI	NI	900210	914804	972855	1403977	50801
	harem_r	1250582	1261660	NI	NI	1207710	1213115	1251027	711660	50801
	roullete	1653996	1648216	NI	NI	1639033	1642805	1642181	1267126	50801
	stochastic	1659796	1653763	NI	NI	1640486	1647711	1651409	1128637	50801
	tournament	1627247	1631481	NI	NI	1638994	1662571	1540997	620452	50801
	OX									
	elitism	953474	982430	NI	NI	749092	755312	948426	140258	50801
	harem_e	961405	962707	NI	NI	759512	747310	959789	126844	50801
	harem_r	1061067	1072773	NI	NI	892213	883368	1056672	111150	50801
	roullete	1659980	1658748	NI	NI	1634679	1657422	1659101	153569	50801
	stochastic	1655048	1654157	NI	NI	1649659	1645656	1640883	206411	50801
	tournament	1624935	1617594	NI	NI	1654932	1647780	1528256	104182	50801
	EAX									
	elitism			-	-	-	-	-	-	7542
	harem_e			-	-	-	-	-	-	7542
	harem_r			-	-	-	-	-	-	7542
	MOX									
	elitism			-	-	-	-	-	-	7542
	harem_e			-	-	-	-	-	-	7542
	harem_r			-	-	-	-	-	-	7542
2103	PMX									
	elitism	2.239E6	2.237E6	NI	NI	2.095E6	2.098E6	2.219E6	2.219E6	80450
	harem_e	2.219E6	2.224E6	NI	NI	2.063E6	2.070E6	2.226E6	2.226E6	80450
	harem_r	2.528E6	2.534E6	NI	NI	2.460E6	2.467E6	2.521E6	2.521E6	80450
	roullete	3.145E6	3.118E6	NI	NI	3.107E6	3.131E6	3.120E6	3.120E6	80450
	stochastic	3.122E6	3.124E6	NI	NI	3.120E6	3.119E6	3.124E6	3.124E6	80450
	tournament	3.110E6	3.108E6	NI	NI	3.117E6	3.099E6	2.974E6	2.974E6	80450
	OX									
	elitism	2.239E6	2.236E6	NI	NI	2.030E6	2.040E6	2.229E6	2.229E6	80450

Continued on NextPage

Continued from Previous Page

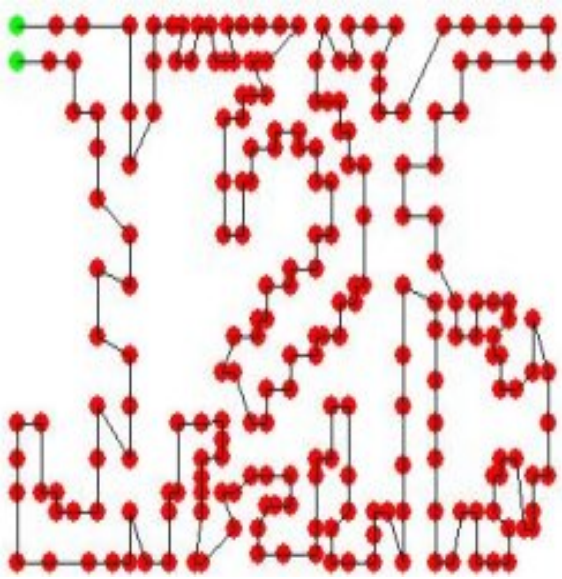
Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	harem_e	2.229E6	2.240E6	NI	NI	2.013E6	2.030E6	2.230E6	2.230E6	80450
	harem_r	2.345E6	2.346E6	NI	NI	2.144E6	2.137E6	2.319E6	2.319E6	80450
	roullete	3.123E6	3.129E6	NI	NI	3.141E6	3.128E6	3.125E6	3.125E6	80450
	stochastic	3.121E6	3.122E6	NI	NI	3.120E6	3.128E6	3.131E6	3.131E6	80450
	tournament	3.100E6	3.091E6	NI	NI	3.127E6	3.113E6	2.945E6	2.945E6	80450
3795	PMX									
	elitism	2.753E6	2.755E6	NI	NI	2.625E6	2.616E6	2.744E6	2.547E5	28772
	harem_e	2.747E6	2.745E6	NI	NI	2.615E6	2.612E6	2.743E6	3.117E5	28772
	harem_r	2.959E6	2.958E6	NI	NI	2.888E6	2.905E6	2.949E6	2.213E5	28772
	roullete	3.487E6	3.483E6	NI	NI	3.490E6	3.487E6	3.476E6	3.220E5	28772
	stochastic	3.463E6	3.483E6	NI	NI	3.471E6	3.479E6	3.473E6	2.533E5	28772
	tournament	3.491E6	3.468E6	NI	NI	3.478E6	3.479E6	3.329E6	1.476E5	28772
	OX									
	elitism	2.756E6	2.750E6	NI	NI	2.619E6	2.605E6	2.751E6	129923	28772
	harem_e	2.754E6	2.747E6	NI	NI	2.595E6	2.614E6	2.749E6	131806	28772
	harem_r	2.843E6	2.827E6	NI	NI	2.682E6	2.686E6	2.837E6	88677	28772
	roullete	3.482E6	3.490E6	NI	NI	3.487E6	3.487E6	3.476E6	125724	28772
	stochastic	3.480E6	3.479E6	NI	NI	3.469E6	3.441E6	3.482E6	300683	28772
	tournament	3.452E6	3.480E6	NI	NI	3.479E6	3.481E6	3.329E6	111090	28772
5915	PMX									
	elitism	3.539E7	3.538E7	NI	NI	3.420E7	3.431E7	3.538E7	3.279E7	565530
	harem_e	3.537E7	3.535E7	NI	NI	3.425E7	3.432E7	3.532E7	3.000E7	565530
	harem_r	3.679E7	3.684E7	NI	NI	3.631E7	3.643E7	3.683E7	3.113E7	565530
	roullete	4.176E7	4.172E7	NI	NI	4.166E7	4.174E7	4.173E7	3.567E7	565530
	stochastic	4.164E7	4.146E7	NI	NI	4.150E7	4.147E7	4.165E7	2.728E7	565530
	tournament	4.158E7	4.133E7	NI	NI	4.162E7	4.168E7	4.053E7	1.635E7	565530
	OX									
	elitism	3.529E7	3.528E7	NI	NI	3.415E7	3.423E7	3.528E7	2534774	565530
	harem_e	3.531E7	3.527E7	NI	NI	3.415E7	3.431E7	3.533E7	2382869	565530
	harem_r	3.606E7	3.605E7	NI	NI	3.495E7	3.496E7	3.605E7	2323640	565530

Continued on NextPage

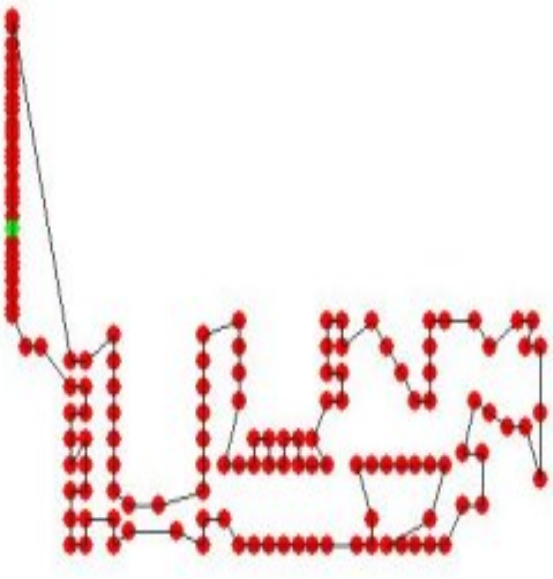
Continued from Previous Page										
Size	Parent Select	AMR	NAMR	SAMR	SNAMR	LP AMR	LP NAMR	AR	GALS	Known Best
	roullete	4.165E7	4.176E7	NI	NI	4.174E7	4.158E7	4.156E7	2257133	565530
	stochastic	4.162E7	4.160E7	NI	NI	4.154E7	4.151E7	4.160E7	2583110	565530
	tournament	4.148E7	4.136E7	NI	NI	4.170E7	4.160E7	4.024E7	2037401	565530
13509	PMX									
	elitism	1.884E9	1.882E9	NI	NI	1.841E9	1.840E9			19982859
	harem_e	1.880E9	1.883E9	NI	NI	1.838E9	1.842E9			19982859
	harem_r	1.923E9	1.922E9	NI	NI	1.909E9	1.906E9			19982859
	roullete	2.117E9	2.118E9	NI	NI	2.122E9	2.116E9			19982859
	stochastic	2.118E9	2.121E9	NI	NI	2.117E9	2.115E9			19982859
	tournament	2.112E9	2.118E9	NI	NI	2.119E9	2.120E9			19982859
	OX									
	elitism	1.881E9	1.881E9	NI	NI	1.843E9	1.843E9			19982859
	harem_e	1.882E9	1.882E9	NI	NI	1.842E9	1.842E9			19982859
	harem_r	1.910E9	1.910E9	NI	NI	1.865E9	1.866E9			19982859
	roullete	2.122E9	2.125E9	NI	NI	2.120E9	2.118E9			19982859
	stochastic	2.119E9	2.116E9	NI	NI	2.117E9	2.113E9			19982859
	tournament	2.108E9	2.097E9	NI	NI	2.118E9	2.115E9			19982859

AppendixB

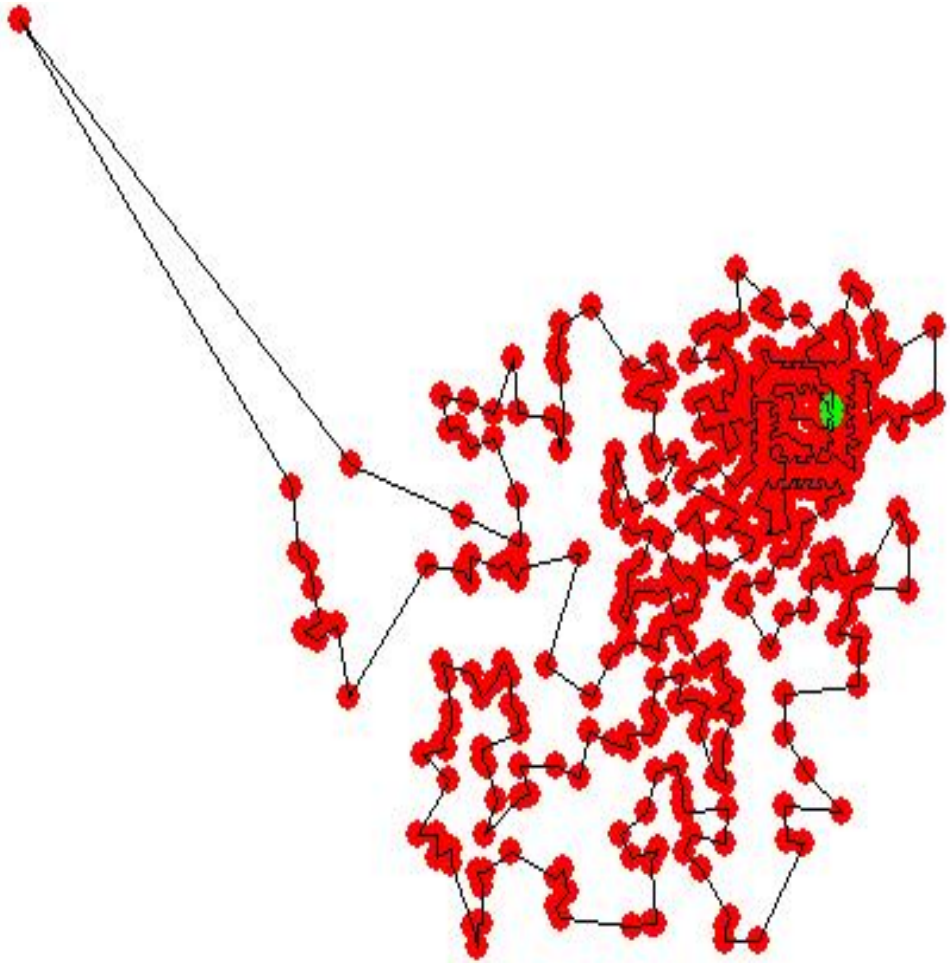
Note: Maps 493 - 2103 all share the same general form: one city is far off in the upper left hand corner and a large clumping for the remaining cities. Because of this, only the 493 city map is displayed. The larger maps (> 2103) are not displayed since the limited space makes it hard to discern any real features of those maps.



(a) 225 City Tour

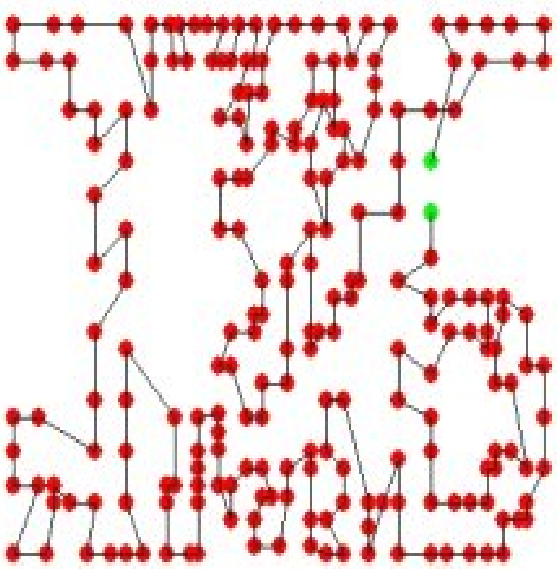


(b) 280 City Tour

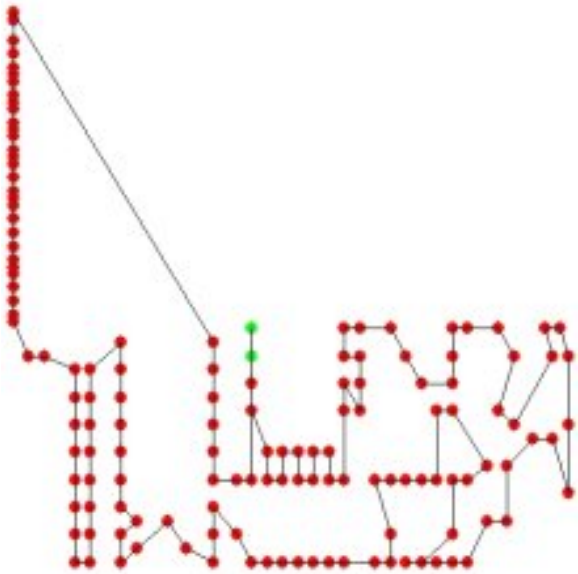


(c) 493 City Tour

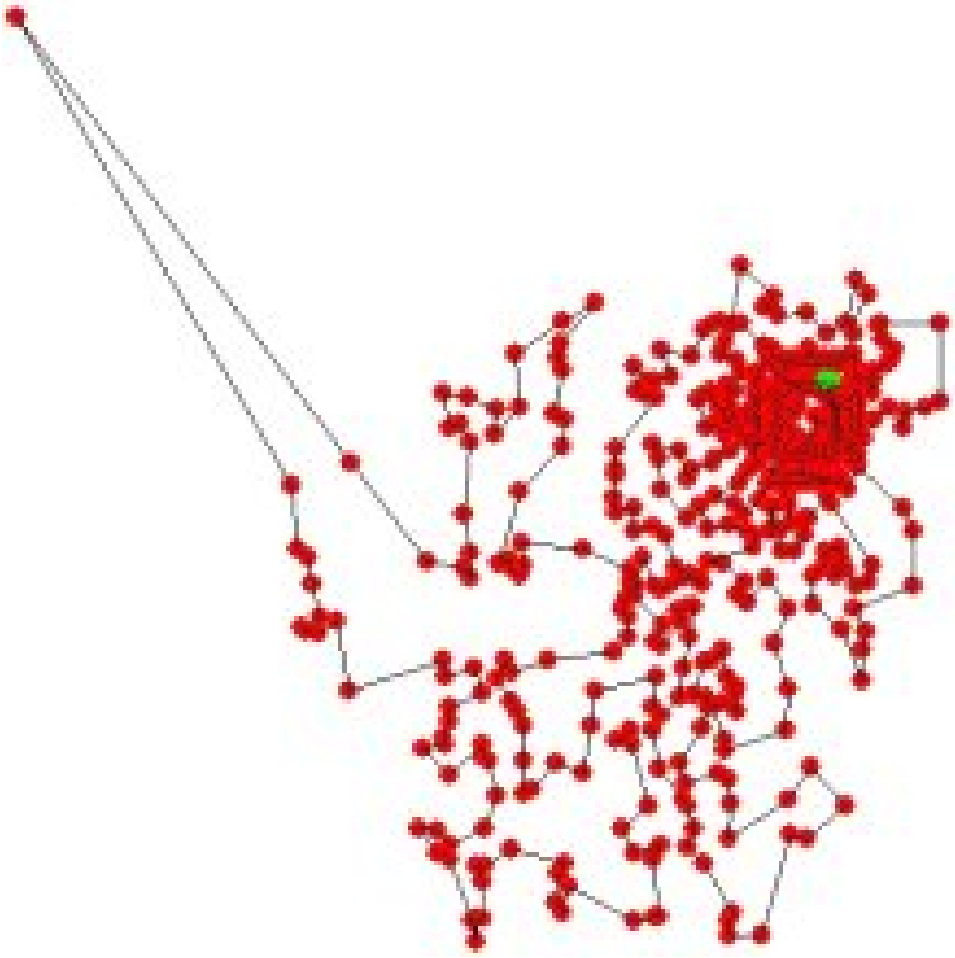
Figure 8: SA Created Tours, Map Sizes 225 - 657



(a) 225 City Tour



(b) 280 City Tour



(c) 493 City Tour

Figure 9: SA Created Tours, Map Sizes 225 - 657