Genetic Algorithms for Applied Path Planning

A Thesis Presented in Partial Fulfillment of the Honors Program

Vincent R. Ragusa

Abstract

Path planning is the computational task of choosing a path through an environment. As a task humans do hundreds of times a day, it may seem that path planning is an easy task, and perhaps naturally suited for a computer to solve. This is not the case however. There are many ways in which NP-Hard problems like path planning can be made easier for computers to solve, but the most significant of these is the use of approximation algorithms. One such approximation algorithm is called a genetic algorithm. Genetic algorithms belong to a an area of computer science called evolutionary computation. The techniques used in evolutionary computation algorithms are modeled after the principles of Darwinian evolution by natural selection. Solutions to the problem are literally bred for their problem solving ability through many generations of selective breeding. The goal of the research presented is to examine the viability of genetic algorithms as a practical solution to the path planning problem. Various modifications to a well known genetic algorithm (NSGA-II) were implemented and tested experimentally to determine if the modification had an effect on the operational efficiency of the algorithm. Two new forms of crossover were implemented with positive results. The notion of mass extinction driving evolution was tested with inconclusive results. A path correction algorithm called make_valid was created which has proven to be extremely powerful. Finally several additional objective functions were tested including a path smoothness measure and an obstacle intrusion measure, the latter showing an enormous positive result.

Department of Computer Science
Under the supervision of Dr. H. David Mathias
Florida Southern College
May 2017

Contents

Acknowledgments

I would like to thank Florida Southern College for financial and other support for this research, 3DR for providing educational pricing and support of open source projects important to research, commercial, and hobbyist projects for MAVs, and Peter Barker for assistance with the drone flight simulator.

I would also like to thank Annie Wu and Vera Kazakova for valuable feedback and collaboration, Isabel Loyd and Susan Serrano for assistance with the statistical analysis, and David Mathias for all of his support and guidance.

1 Introduction

The very idea of efficiency is born from asking "Why is this task accomplished in this way?" and following that question with "Is there a better way?". This way of arriving at efficiency leaves a lot of room to interpret "better" and as such the task of finding "the best" of anything can be extremely complicated. In mathematics and in computer science the task of finding the best of something is called an optimization problem.

1.1 Optimization Problems

Optimization problems can be thought of as search problems where the task is to search for the optimal solution to a problem from among a pool of candidate solutions. There are typically many candidate solutions, each, usually, with an associated value (or values) which are desired to either be maxamized or minimized[1]. It is convenient to call all optimization problems minimization problems because any value you wish to maximize can be multiplied by (-1) and then minimized to achieve the same effect. Also note that it is common for there to be more than one candidate solution which achieves the optimal value[1].

1.1.1 Single-Objective Optimization

Single-objective optimization problems seek the minimum of a single value. Single-objective problems are typically solved by mathematical methods such as evaluating derivatives and gradient descent as well as basic computational methods like greedy strategies. For example, consider a single-objective optimization problem where the goal is to minimize the potential energy of a spring $U(x)$.

$$U(x) = \frac{1}{2}kx^2$$

Evaluating $\frac{dU}{dx} = 0$,

$$\frac{d}{dx}U(x) = kx = 0$$
$$\Rightarrow x = 0$$

it is seen that there is a critical point when $x = 0$. Furthermore evaluating $\frac{d^2U}{dx^2}$ at the critical point,

$$\left.\frac{d^2U}{dx^2}\right|_{x=0} = k$$
$$k > 0$$

reveals that the critical point is a minimum because the second derivative is positive at the critical point. Therefore the minimum of $U(x)$ is at the point $x = 0$, or when the spring is neither compressed or stretched. Mathematical methods are often the easiest method for solving single objective optimization problems.

1.1.2 Multi-Objective Optimization

Multi-objective optimization problems seek to minimize n values simultaneously. Under ideal circumstances each objective is independent and the problem reduces to solving n single-objective optimization problems in parallel. This is almost never the case however as a single property of a candidate solution usually affects multiple objective values. Consider the task of maximizing the volume of a box (minimizing the volume multiplied by (-1)) and minimizing the surface area of the box. Note $x, y, z \in (0, \infty)$.

$$V(x, y, z) = -xyz$$

$$A(x, y, z) = 2(xy + xz + yz)$$

If an attempt is made to minimize $V(x, y, z)$ using the same mathematical method used for the spring example in section 1.1.1:

$$\nabla V = -(yz, xz, xy) = (0, 0, 0)$$

$$\Rightarrow (x = y = 0)\ OR\ (x = z = 0)\ OR\ (y = z = 0)$$

a contradiction is reached because x, y, and z cannot equal zero. A second attempt, this time to minimize $A(x, y, z)$ reveals:

$$\nabla A = 2(y + z, x + z, x + y) = (0, 0, 0)$$

$$\Rightarrow (y + z = 0)\ AND\ (x + z = 0)\ AND\ (x + y = 0)$$

Solving by elimination:

$$(x, y, z) = (0, 0, 0)$$

Again a contradiction is reached because x, y, and z cannot equal zero. Finally, let $R(x, y, z)$ be defined as the ratio of the area to volume:

$$R(x, y, z) = \frac{A(x, y, z)}{V(x, y, z)} = \frac{2(xy + xz + yz)}{xyz} = \frac{2}{z} + \frac{2}{y} + \frac{2}{x}$$

Searching for critical points reveals:

$$\nabla R = -2(\frac{1}{z^2}, \frac{1}{y^2}, \frac{1}{x^2}) = (0, 0, 0)$$

which has no solutions. It seems that without some new mathematical or computational tools these problems cannot be solved.

Scalarization

Sometimes it is still useful to attempt to optimize a multi-objective problem by using a method known as *scalarization*. Scalarization is simply the transformation of n objective functions $\{f_1, \cdots, f_n\}$ into a single function F via a weighted sum[2], where $c_i \neq 0$ is the weight on f_i.

$$F(p) = \sum_{i=1}^{n} c_i f_i(p)$$

Often, the weights sum to 1 so that each weight represents a percentage of the final sum.

$$\sum_{i=1}^{n} c_i = 1$$

Individual objectives can be given preference by giving them larger weights. Raising the objective function to a different power $k \neq 0$ can be used to emphasize ($k > 0$) or de-emphasize ($k < 0$) changes in an objective function.

$$F(p) = \sum_{i=1}^{n} c_i (f_i(p))^{k_i}$$

The biggest drawback of scalarization is that contributions by individual fitness measures are anonymous, hiding useful data about the search space[2]. Additionally, it falls victim to things like change blindness, or when multiple fitness functions change but the weighted sum has a net change of zero. Scalarization is often abandoned for better methods of dealing with multiple objectives simultaneously.

Pareto Optimization

One popular method of handling multiple objectives in an optimization problem is to rank them not via the fitness values directly, but by their relative *Pareto efficiency*. Simply put, a candidate solution to a multi-objective optimization problem is Pareto efficient (or Pareto optimal) if it cannot make any progress in one of its objectives without causing negative changes in some or all of the other objectives[3].

This is more easily illustrated by first defining a vector comparison operator "$\prec$". Deb *et al*[4] define this operator such that vector $\mathbf{v}_1$ is partially less than vector $\mathbf{v}_2$ when:

$$\mathbf{v}_1 \prec \mathbf{v}_2 \equiv (\forall i \mid \mathbf{v}_{2,i} \geq \mathbf{v}_{1,i}) \wedge (\exists i \mid \mathbf{v}_{1,i} < \mathbf{v}_{2,i})$$

With this, comparing two solutions is done by recognizing each objective's fitness score as a component of a fitness vector with dimension equal to the number of objectives. If the fitness vector of one solution is partially less than that of another then the former *dominates* the latter.

With this new operator *Pareto optimal* can be defined as a solution that cannot be dominated by another solution. Note that there can be (and often are) an infinite number of Pareto optimal solutions to a given multi-objective optimization problem. This method of ranking solutions is therefore not capable of discerning one true optimal solution, but the set of non-dominated solutions. It is left up to another sorting/ranking algorithm or human expert to choose from the set of Pareto optimal solutions the best one to solve the given problem.

1.1.3 Computational Complexity

Computational complexity theory is the study of the *intrinsic complexity* of computational problems[5]. That is to say, given a sufficiently defined computational problem, computational complexity theory aims to make concrete statements about the minimum, maximum,

and average computational effort needed to solve the problem, and the amount of computer memory required to do the computation.

Computational complexity theory does not deal directly with optimization problems because their open ended nautre makes their analysis dificult. Rather the field works entirely with decision problems[6, 5]. It has been shown, however, that optimization problems can be transformed into decision problems and therefore it is possible to study optomization problems based on the study of their counterparts [6, 5, 1].

The *time complexity* of an algorithm is the number of atomic computational operations needed for that algorithm to halt on the given input as a function of the length of the input[7]. There are some simplifying assumptions that are made when studying time complexity, most significant of which is the use of $O()$ (pronounced "big-oh") notation. $O()$ notation describes any arbitrary function as being *asymtotically equivelant* to a simpler function[1]. This relationship comes from the fact that higher degree terms in a function's definition will dominate the lower order terms as the input of the function grows very large. Therefore the function can be approximated by its highest degree term alone, if the input values are assumed to be very large. For example $f(x) = 2x^3 + 7x - 5$ would be rewritten as $O(f(x)) = O(n^3)$. This gives us a convenient way to compare how the time complexity grows as n (the length of the input) grows. For example, an algorithm that can solve a problem in $O(n^2)$ time is asymptotically better than an algorithm that solves the same problem in $O(n^3)$ time.

Classifying problems based on the time complexity of the best algorithm for solving a problem sheds light on the problem's intrinsic difficulty. Alphabetizing a bookshelf, for example, has the time complexity $O(n\ log(n))$ which puts it in the class P (see Figure 1), which stands for "polynomial time". Surprisingly, research has shown that there are some problems for which finding a solution is practically impossible. Generally speaking, these kinds of problems have a time complexity of at best $O(2^n)$. These problems belong to the time complexity class called EXP (or EXPTIME), which stands for "exponential time". Problems like playing chess are in EXP. An interesting subset of EXP is the complexity class NP which stands for "non-deterministic polynomial time". Although it is an open problem in complexity theory if P=NP or if P≠NP, it is generally assumed that P≠NP. A consequence of this assumption is that problems in NP might take $O(2^n)$ time to find the correct solution. What distinguishes NP from EXP is that checking an arbitrary solution for correctness only takes polynomial time, placing the problem of verifying the solution in the class P while the overall problem is intractable.

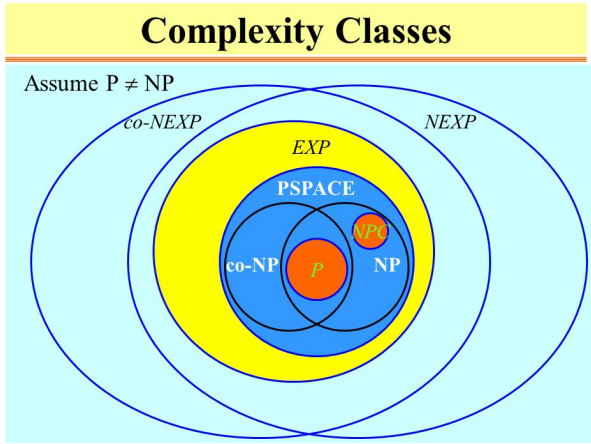


Figure 1: This diagram shows the relationship among various complexity classes (assuming P $\neq$ NP). P, all problems solvable in polynomial time, is contained inside NP, all problems checkable in polynomial time, which itself is contained inside EXP, all problems which have a runtime of at least 2^N.

The most intuitive example of a problem in NP is the Sudoku puzzle (Figure 2). Given a sparsely filled 9x9 matrix, fill each remaining element such that every row, column, and 3x3 region contains the numbers 1 through 9 exactly once.

6				2				9
	1		3		7		5	
		3				1		
	9						2	
2			8	7	5			3
		5		1		4		
	7			8			9	
		1		4		8		
			2	5	9			

Figure 2: Sudoku puzzles are perhaps the most widely recognized NP-Complete problem. The difficulty of the puzzle arises from the computational complexity inherent in such problems where trial and error is the only real strategy available.

This might seem at first to be nonsense, that Sudoku is practically impossible to solve, seeing as many people solve Sudoku puzzles in the morning paper with relative ease. How can Sudoku be "practically impossible"? Recall that time complexity is a function of the size of the input. A Sudoku puzzle is actually a special case of a mathematical object called a *Gerechte Design*[8]. A Gerechte Design of order n is an nxn matrix, partitioned into n regions each containing n cells, such that the symbols $1, \cdots, n$ appear exactly once in each row, column, and region. Therefore a Sudoku grid is simply a Gerechte design of order 9. Furthermore the time complexity of finding valid Gerechte designs of a given order is a function of the order itself which is asymptotically equivalent to $O(2^n)$, so a 115x115 Sudoku

grid would take the worlds fastest super computer the age of the universe to solve[1]. In real world NP and EXP problems, like protein folding and path planning, the size of the input can grow well beyond 115 for even the simplest cases, and access to super computing capabilities is limited or too expensive. For all practical purposes, solving these kinds of problems is simply impossible.

1.2 Path Planning

Path planning is a broad class of optimization problems where the task is to find a viable route from a source location to a destination within a static environment[9]. It was discussed in Section 1.1.3 that path finding is a"hard" problem. It belongs to the complexity class PSPACE-Hard (see Figure 1) which makes it at least as hard as all other problems inside PSPACE[2], but potentially as hard as a problem in any of the more dificult complexity classes. The primary algorithmic objective is to optimize some quantity, most often path length. In real world situations, however, additional objectives may also need to be taken into account. Examples include minimizing the complexity of the path by reducing the number of turns or the sum of the angles of the turns[10, 11, 12, 13, 14], maximizing the safety of the resulting path[10, 12], and maintaining a minimum or maximum distance from obstacles or the terrain[11, 13, 14]. Thus, path planning is often a multi-objective optimization problem. An instance of the path planning problem considered here consists of a starting location, s; a destination location, d; a (possibly empty) list of obstacles, obs; and a (possibly empty) list of intermediate targets, $goals$ in a largely unconstrained continuous environment. s and d are represented by GPS coordinates and can exist anywhere in the environment. Obstacles and intermediate goals can require flight in any direction and co-location of the s and d can require a circuitous path. Obstacles are circular or polygonal. If obstacle o_i is circular, it is represented by the GPS coordinates of its center and a radius in meters. If polygonal, it is represented by a list of vertices, each of which is a GPS location. Intermediate targets are locations to which the vehicle must fly, prior to the destination, to successfully complete its mission. These may represent locations near RF-enabled sensors or from which a photograph must be taken, for example. Each element of $goals$ is an appropriately sized circle represented in the same format as a circular obstacle.

A solution to the problem consists of a sequence of *waypoints*,

$$S = [w_0, \ldots, w_n]$$

where $w_0 = s$ and $w_n = d$. Each waypoint w_i is represented as a real-valued pair, (lat, lon), consisting of the latitude and longitude of the location. In general, waypoints are points of articulation in the path, though it is possible for a waypoint to subdivide a straight path segment. A solution is considered valid if the path defined by S avoids all obstacles in obs and reaches all targets in $goals$.

[1]As of March 2017 the worlds fastest super computer is the Sunway TaihuLight housed in the National Super-computing Center in Wuxi China. With a top measured performance of $93.015 * 10^{15}$ FLOPS it can do $\simeq 4.02 * 10^{34}$ calculations in $4.32 * 10^{17}$ seconds ($13.7 * 10^9$ years), the estimated age of the universe. Given exponential time complexity, a problem size of 115 will require $2^{115} \simeq 4.02 * 10^{34}$ calculations.

[2]PSPACE is the space complexity class that contains all problems that require, at most, polynomial memory to solve. As with time complexity classes, the space complexity is related to the size of the input, n. Figure 1 shows PSPACE in relation to the previously discussed time complexity classes.

1.3 UAVs and MAVs

An unmanned aerial vehicle, commonly abbreviated UAV, is exactly what the name implies; UAVs are aircraft which are piloted *remotely* by a human or a computer. The term UAV encompasses a wide variety of aircraft including stealth military airplanes and $20.00 toys for kids. Micro aerial vehicles, abbreviated MAV, are small UAVs, the kind typically for sale in a hobby store, or used in film to capture footage from high above.

Today, most UAVs are piloted by humans, with the high budget and military grade pilots having access to computer assistance and auto-pilot. It is an open area of research to give full autonomy to UAVs. Because MAVs are relatively inexpensive and easy to work with, most applied research is done with them.

One area of autonomous UAV research is in path planning. Being fully autonomous means the UAV needs to be able to receive instructions for a mission and be able to execute that mission in whichever way is optimal, including choosing the optimal path to fly during the mission. For this reason, path planning algorithms are a key part of UAV autonomy research.

Consider a MAV that is tasked with transporting a package from the post office to a residential address. The drone must consider the most efficient rout to the address and back. It must also ensure it steers clear of any no fly zones like airports. It must also avoid tall buildings, construction equipment, trees, and potentially other drones as it flies. As mentioned in the previous section, path planning is a multi-objective optimization problem, which makes it difficult to solve perfectly. Additionally, if the computations are to take place on-board the drone, energy consumption related to computation must be kept to a minimum, since the battery powering the computer also powers the motors.

1.4 Genetic Algorithms

Approximation algorithms are typically used when finding the solution to a problem known to be NP-Hard or worse in time complexity. The extreme cost in computation time is usually incurred because the algorithm is designed to make perfect decisions, which in turn requires perfect information, which takes time for the algorithm to gather and analyze. In these circumstances an algorithm that makes imperfect decisions during the search for a solution might still come very close to the global optimum without performing as many computations. The output of such an algorithm, then, is not the optimal solution but rather an approximation of the optimal solution, and the algorithm used to find it is an approximation algorithm. While some approximation algorithms are written to solve specific problems, others are designed to operate on optimization problems in general.

In 1962 John H. Holland published a paper[15] that laid the groundwork for an entire field of computer science known as evolutionary computation. His key idea was to design algorithms that could adapt to the problem they are solving, similar to how organisms in nature evolve to better survive in their environments. There are a wide variety of ways to design such a system, but the one Holland became famous for designing is known today as the *simple genetic algorithm*[2].

Genetic algorithms use randomization and the principles of evolution by natural selection to *evolve* or *breed* good approximate solutions. The genetic algorithm leverages the fact that

there are many candidate solutions in a search space to converge towards the optimal solution faster. It does this by maintaining a population of candidate solutions. By applying a mate selection algorithm, two candidate solutions can be breed together producing a child with similar characteristics. This is usually done many times until a new population of children has been formed. Each new set of children is called a *generation*. The process is repeated, using the children as the parents for the next generation. By ranking each candidate solution in the population with respect to the optimization objective(s) as discussed in section 1.1 and using a mate selection algorithm that gives preference to solutions with a better ranking, each subsequent generation will contain members that are slightly better on average than the previous generations. Over a long sequence of iterations one could say the population has evolved solutions that are good approximations of the optimal solution.

1.4.1 Solution Encoding Scheme

Part of what makes genetic algorithms unique among evolutionary computation models is how the population of solutions is encoded and operated on. This is where the genetic algorithm takes inspiration from nature. Each solution is encoded as a sequence of bits, values, or symbols that can be decoded back into a solution for evaluation against a fitness function. This is analogous to how each organism's DNA is a sequence of molecules that can be decoded into a set of recipes and instructions for building the organism. Each organism is then "evaluated" by its environment to determine its fitness. For example the sequence $[1, 0, 0, 1, 0, 1, 0]$ could be interpreted as 74 in base 2, a sequence of left or right turns (if 0 maps to left and 1 maps to right then the sequence is $[right, left, left, right, left, right, left]$), or any number of alternate interpretations. Likewise the sequence $[1.1, 2.5, 7.3, 6.7]$ could be two points on an XY-Plane (if alternating points correspond to x and y respectively) or the setting of 4 dials on a machine that can turn from 0 to 10. Part of implementing a genetic algorithm is to find an encoding scheme by which you can transform a candidate solution into a sequence of characters and back again.

1.4.2 Genetic Operators

The genetic algorithm's main method of searching the solution space for the optimum is through the application of *genetic operators* to members of the population of candidate solutions. Extending the biological analogy, the traditional genetic operators mimic the ways in which DNA is changed over time as members of a species mate and produce children.

Crossover

Crossover is the first traditional genetic operator. In nature, when two organisms mate, each parent contributes half of the needed chromosomes to the child. They are not, however, simply internalized as the child's own chromosomes. Instead the two chromosomes come apart, trade segments of DNA with each other, and then recombine into two chromosomes that are each distinct from the ones the parents contributed. This is why a child looks similar to, but is not an exact clone of, their parents.

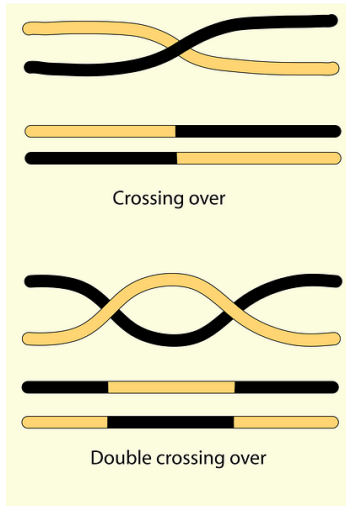


Figure 3: This cartoon demonstrates the concept of crossover.

The crossover operator in a genetic algorithm works in exactly the same way as the biological crossover event. Consider two parent solutions, P_1 and P_2, represented as

$$P_1 : [1, 0, 1, 0, 0]$$

$$P_2 : [0, 1, 0, 0, 1]$$

The crossover operator first randomly chooses a *cut point* where each chromosome will be separated into two pieces.

$$P_1 : [1, 0|1, 0, 0]$$

$$P_{1[1]} : [1, 0]$$

$$P_{1[2]} : [1, 0, 0]$$

$$P_2 : [0, 1, 0|0, 1]$$

$$P_{2[1]} : [0, 1, 0]$$

$$P_{2[2]} : [0, 1]$$

Two new solutions, C_1 and C_2, are then created by recombining these pieces together, typically putting together $P_{1[1]}$ with $P_{2[2]}$ and $P_{2[1]}$ with $P_{1[2]}$.

$$C_1 : [1, 0, 0, 1]$$

$$C_2 : [0, 1, 0, 1, 0, 0]$$

Notice that in general, the children produced this way will have different lengths than the parents unless the same cut point is used for both parents. Whether or not the length of the sequence can change is entirely problem dependent. For some chromosome encoding schemes, changing the length of a chromosome might corrupt the data or cause run-time errors in the algorithm. Also note that crossover is not limited to a single cut point. The use

of more than one cut point (up to a maximum of $\ell - 1$ where ℓ is the length of the parent) is not uncommon. The naming convention is usually single-point crossover, double-point crossover, n-point crossover, and uniform-crossover ($n = \ell - 1$).

The form of crossover presented here is the most common kind, and the most common modifications of it. There are however specialized kinds of crossover that are used for specific problems like the traveling salesman problem. The variations of crossover are virtually unlimited.

The function of the crossover operator is to serve as an *exploitation* mechanism. Consider each candidate solution in the population as a point in some ℓ-dimensional space. Each member of the population is connected to each other point in this space to form a lattice via a transformation by the crossover operator[2]. The crossover operator then lets the genetic algorithm travel along the connections in the lattice in order to find the point in the lattice that is closest to the optimum point of the space. In other terms, crossover is searching for the optimal combination of the values which already exist in the population somewhere. It cannot create new values in any sequence it operates on.

Mutation

Mutation is the other traditional genetic operator. Like in biology, whenever DNA is operated on mistakes can occur. These mistakes are called mutations. Similarly, in genetic algorithms mistakes are engineered to happen. Consider the parent solution P_3,

$$P_3 : [A, B, c, d, K, v]$$

the mutation operator will randomly select a position and change its value to create a child, C_3.

$$C_3 : [A, B, \mathbf{G}, d, K, v]$$

Mutation can be implemented in any number of ways. If the sequence is composed of 1's and 0's then a mutation might be to just turn one into the other. If the solution is encoded as a sequence of floating point values, mutation might add or subtract a small value from the original. If the encoding is allowed to have a variable length, then an add or delete mutation could lengthen or shorten the sequence, respectively. It is common to have many forms of mutation, and select one randomly each time a mutation event occurs.

If crossover was the exploitation operator, mutation is the *exploration* operator. Because mutation can alter the values within the sequences it operates on, it is effectively adding new points to the lattice of solutions in the solution space[2]. This in turn lets the crossover operator travel to new locations in the search space which were previously inaccessible.

1.4.3 Selection and Niching

Another important factor in the behavior of genetic algorithms is the method of choosing who will be operated on by the genetic operators. How this is accomplished will have a large effect on the search process. Just as dog breeders are extremely cautious to only mate their dogs with partners of quality pedigree, so too should a genetic algorithm be picky. Applying crossover to two parents selected randomly will produce, on average, a population of average

individuals. Applying crossover to two parents who are known to be good solutions will, on average, produce solutions above average. Each generation will push ever closer to the global optimum. There are two key aspects to this process: being able to tell which solutions are good, and how to restrict selection to those good solutions.

Fitness Functions

As discussed in section 1.1 there are many ways to attempt to rank candidate solutions, p_i, with respect to each other. After deciding on a ranking mechanism, called a *fitness function*, $f(p_i)$, in genetic algorithm literature, the population of candidate solutions can be ranked and sorted so that individuals with higher fitness may be identified.

Selection

The actual selection of parents can be done in many ways. The goal is to relate the probability of selection to the fitness of the individual. One such method, sometimes called wagon wheel selection, roulette wheel selection, or dart board selection, sums together the total fitness of the population

$$F = \sum_{i=1}^{n} f(p_i)$$

and then assign probability of selection as a direct consequence of how much each member contributed to that sum.

$$P_i = f(p_i)/F$$

Next each individual is given a selection threshold T_i, which is the individuals probability of selection plus the threshold of the previous $(i - 1\text{st})$ member.

$$T_0 = P_0$$

$$(i > 0), T_i = P_i + T_{i-1}$$

When it is necessary to choose a parent, selection is accomplished by generating a random number *Rand* between 0 and 1 and choosing the first individual who has a threshold greater than *Rand* by examining the individuals in order of increasing threshold values. This can be visualized by dividing a circle with area 1 into wedges of area P_i. By throwing a dart at the circle, a wedge is randomly selected and the probability of hitting a larger wedge is higher than hitting a smaller wedge. The other names for this method come from alternate images of this process, envisioning the circle as a roulette wheel and selection as the ball, or envisioning the circle as a wagon wheel, and shooting an arrow between the spokes as selection.

Another popular selection method is called tournament selection. A preliminary selection occurs in which n members of the population are chosen with equal probability. From these n, the one with the best fitness is selected as the winner of the *tournament*. If two parents are needed, as in the case of a crossover event, a second tournament is run to select the second parent. The number of individuals n who are chosen in the preliminary stage is known as the *tournament pressure*, and is a parameter that can be tuned to achieve varying

degrees of strictness in the selection process. For example when $n = 2$ (also called a *binary tournament*), the tournament pressure is said to be low, and individuals with low fitness will more frequently be allowed to move on as parents. As the tournament pressure increases it becomes more challenging for less fit members of the population to move on as parents. Also note that a selection pressure of $n = 1$ is simply random selection.

Elitism

In most genetic algorithm implementations once a child population is created from a parent population, it is used as the parent population for the next generation. There is a subtlety to this procedure that often leads to the algorithm taking much more time than is necessary to converge to a good solution. The subtlety lies in the fact that crossover and mutation are not *guaranteed* to produce a result better than the solutions that are used as their input.

Actually, the most common result of mutation and crossover is complete garbage. The genetic algorithm is relying on the selection function to defy these odds and arrive at better solutions than random chance alone could produce. However, because mutation and crossover are not monotonically increasing the fitness of the solutions they operate on, it is entirely likely that the resulting child population of a given generation could be holistically worse than the parents. In essence, the genetic algorithm is deleting good solutions on accident when it throws away the parent population.

The solution to this problem comes as an additional complication to the algorithm called *elitism*. Elitism is the practice of preserving the best member (or best n members) of the parent population each generation. There are a wide variety of ways to implement elitism in a genetic algorithm.

One method, used in NSGA-II [16], is to combine the parent and child populations together into a mixed population. From this mixed population, only the top 50% of the solutions are sent on as the parents for the next generation. This method is simple, and effective. It also has the added benefit of not requiring any additional housekeeping as other methods might require.

Another method of elitism, often used in multi-objective genetic algorithms, is a method called *archiving*. Archiving methods of elitism copy the best members of the population into a secondary population, called the "archive", but still allow these good solutions to "die out" as they would without any elitism. The advantage with archiving is that the elitism does not interfere with the normal process of the genetic algorithm, artificially preserving any member longer than the selection function would have allowed it.

Some archiving methods disallow archived solutions from being allowed to participate in the creation of children at any point. Still others allow them to participate as if they were in the parent population, or allow them to participate only every x generations.

There are different methods of maintaining the archive as well. Some implementations do not consider the members already in the archive when adding a new member, while others will only permit a new solution to be archived if it is deemed to be good enough. This decision is usually made by testing if it is better than any member already inside. In these implementations, it is also common to remove members from the archive that have been shown to be less fit than a newly archived solution.

Niching and Population Diversity

One potential pitfall of genetic algorithms is the potential to get stuck on a local optimum. In genetic algorithms converging to an optimum (local or global) manifests as a decrease in the diversity of the candidate solutions in the population as they each become like one another (via crossover).

In order to prevent premature convergence and to help facilitate exploration of the search space, many genetic algorithms implement a niching mechanism. In ecology, a niche is the role and position a species has in its environment. In genetic algorithm literature, the term is borrowed and similarly refers to how specialized the members of the population are. Similar to how a fitness function ranks each member with respect to the optimization problem, a niching algorithm is used to measure how unique an individual is. To avoid premature convergence, and thus prevent getting stuck on a local optimum, this niching score is factored into the selection process. In tournament selection where non-domination fronts are used as the primary fitness measure, for example, a niching function can be used to break ties. The net effect of a niching algorithm should be to preserve the population's most unique members from generation to generation because their fitness is augmented by the niching score. This allows crossover and mutation to continue to make progress at the fringes of the solution space, and potentially find a new, better, optimum.

1.4.4 Tuning a Genetic Algorithm

The biggest challenge when using a genetic algorithm to solve an optimization problem is configuring all of the degrees of freedom to give the most efficient problem solving algorithm.

For example there needs to be a balance between crossover and mutation to achieve a balance between exploration and exploitation. Too much exploration, and the algorithm will simply ignore any optimum it finds. Too much exploitation, and the algorithm will latch onto the first optimum it finds.

There needs to be a balance in the selection process too. Balancing selection pressure with niching is key in ensuring both speedy convergence, and enough genetic diversity to find the correct optimum.

2 Problem Statement

Path planning for autonomous UAV flight is a computationally expensive task. In order to reduce the energy and time costs of planning paths on-board a UAV (i.e. without a connection to a centralized control center), genetic algorithms are proposed because of their utility as multi-objective optimum approximation algorithms.

For a genetic algorithm to be considered a viable solution to the path planning problem proposed in this thesis it must accomplish a minimum set of requirements. For a given environment, the genetic algorithm will be required to:

- For test maps:
  - Produce a path which is free of obstacle collisions.

  - Produce a path with length $\leq$5% of the best known length.

- For unknown maps:
  - Produce a path which is free of obstacle collisions within three attempts.
  - Take no longer than 2 minutes for any attempt to find a valid path.

- Visit all (if any) intermediate goal locations, in any order.

- Output the final path in GPS co-ordinates for compatibility with the drone's software.

3 The Algorithm

The first step in determining if genetic algorithms are well suited to the path planning problem is to implement a genetic algorithm that meets the most basic requirements defined in the problem statement. After implementing a genetic algorithm, based on the well known NSGA-II[16], a baseline was established for how well the genetic algorithm could solve the path planning problem, by engineering inputs of increasing difficulty until a point of failure was identified.

3.1 Pathfinder

Here the details of the genetic algorithm build for this work, called "Pathfinder", is presented.

3.1.1 Genome & Solution Encoding

In the original genetic algorithm, the chromosome was treated as a bit string and genetic operators are applied at the bit level. For pathfinding, numerous encodings have been considered. Hermanu *et al* [17], Sedighi *et al* [18], and Ahmed and Deb [10] have reported on the effectiveness of different chromosomes. Directly applying these results to the problem is difficult due to the discrete environments they each consider. Zheng *et al* [14] work with a continuous environment. Their chromosome consists of a sequence of waypoints, each with a state variable indicating feasibility of the incident path segments. Similarly, the chromosome used here consists of a sequence of waypoints S , in which each waypoint is represented as a pair of real values, interpreted as GPS degrees of latitude and longitude. State variable, like those mentioned above, are not used in pathfinder. Rather path validity is measured during child evaluation and used as an objective function. The chromosome used here is of variable length so as not to limit the degrees of freedom like numbers of turns and to move away from enforced monotonicity like those used in the works cited above.

In Pathfinder, operators are applied to waypoints or waypoint components, as opposed to bits. The reason for this is to constrain the results produced by the operators. For example, in a binary encoding scheme flipping a single high-order bit in the chromosome could result in a huge change in the position of a waypoint. Because the algorithm is designed to be used for navigation on an actual vehicle with very limited flight time, such large-scale changes are impractical. No waypoint is viable if it requires movement to a position beyond what

is possible within the limits of the battery life of the vehicle. Following a similar argument, crossover was not implemented in the first version of Pathfinder.

3.1.2 Objectives

The objectives to optimize are: path length, number of obstacles hit, number of targets hit and smoothness [12]. Smoothness, defined here as

$$\sum_{i=0}^{n-2} |(\measuredangle w_i w_{i+1} w_{i+2}) - \pi|$$

(radians) is related to the sum of the angles between pairs of consecutive path segments. Minimizing this quantity reduces sharp turns which require more time and, possibly, more maneuvering for certain types of vehicles. In order to be able to minimize all objectives, as discussed in 1.1, the negative number of targets hit is used as an objective function.

3.1.3 Algorithm Structure

The initial population is seeded during "generation 0" which is where all of the intiialization for the algorithm occurs. The initial parent population p_0^P, consists of randomly generated members. The chromosome for each random member contains between 1 and 5 randomly generated waypoints. Each of these waypoints must be within some threshold distance of the starting point, to keep the search reasonably focused. The seminal work of Deb *et al*, NSGA-II [16], was used as a starting point for Pathfinder. It was chosen because it is by far the most prominent for multi-objective optimization and is reasonably easy to implement. NSGA-II requires that a population of children exists before each new generation. Therefore during generation 0 a child population, p_0^C, is also randomly seeded which in later generations will contain the resulting output of the genetic crossover and mutation operators. Pathfinder only applies mutation and selection operators to maintain a population of candidates. The population is sorted into domination fronts as described in the Pareto optimization discussion in Section 1.1. The base algorithm serves as a control in each of the presented experiments.

From generation g_j, generation g_{j+1} is created as follows: Let generation p_j^P be the parent population. First a child population p_j^C is created such that $|p_j^P| = |p_j^C|$. Then p_j^P and p_j^C are merged into a combined population $p_j^{P \cup C}$. $p_j^{P \cup C}$ is then sorted into domination fronts. Within a front, a niching measure [19] known as *crowding distance* [16] is used to impose a partial order on the members. The crowding distance of a member is a measure of the dissimilarity of that member to other members of the front. Greater values are preferred to help ensure diversity in the gene pool. After $p_j^{P \cup C}$ is sorted, it must be cut down to the original size of p_j^P for use as the parent population in generation. Let k be the largest index such that

$$\sum_{i=0}^{k} |f_i| = y \ \leq \ n = |p_j^P|$$

where each f_i is a domination front, with increasing values of k representing further distance from the Pareto front. Then p_{j+1}^P consists of all members from f_0 through f_k plus an additional $n - y$ members from f_{k+1} chosen by maximum crowding distance.

Child creation occurs via the mutation of a member of p_j^P. To select a member to mutate, tournament selection is used with a tournament pressure of 4. When the best member of the 4 has been chosen, it is mutated.

3.1.4 Genetic Operators

Four mutation operators are implemented: *add*, *delete*, *swap*, and *move*. When a member is chosen for mutation, exactly one of the operators is applied. The probabilities of the operators are 0.1, 0.05, 0.1 and 0.75, respectively. While *add* inserts a new waypoint into S, the other operators each alter an existing waypoint. Distinguished waypoints $w_0 = s$ and $w_n = d$ cannot be mutated.

The *delete* operator removes randomly selected waypoint w_i from S for some $0 < i < n$. The *swap* operator randomly selects a pair of adjacent waypoints, w_i and w_{i+1} $0 < i < i+1 < n$, and exchanges their positions in S. The *add* operator works by randomly selecting a pair of adjacent waypoints w_i and w_{i+1} , $0 \leq i < n$ and creates a new waypoint, w_k, at the midpoint of the path segment between them. The *move* operator is then applied to w_k so that the new point is not simply the bisecting point, which would strictly decrease the fitness of the path.

The *move* operator is somewhat more complex. A waypoint, w_i, $0 < i < n$, is chosen at random. The latitude and longitude values of w_i are changed independently according to a Gaussian distribution with mean 0 and standard deviation σ. The value for σ is selected by fair coin flip from among two candidates, *small move* and *big move*. *small move* is fixed at 0.00002 degrees(GPS), approximately 2 meters. The value of *big move* depends on the problem instance. For instances with small obstacles, it is 0.0002 degrees ($\approx$ 20 m) and for instances with large obstacles, it is 0.00067 degrees ($\approx$ 69 m). The motivation for using variable mutation size comes from empirical experience when running the algorithm. It is clear that one size does not fit all input cases, and as a compromise between doing an in depth study on the optimal mutation size and ignoring the empirical data, the choice was made to manually set the mutation size to a more suitable value for those inputs that had need of it. In practice, this change simply has the effect of scaling up the size of mutation to match the scale of the obstacles in the input, so that they are within the same order of magnitude of each other.

4 Experiments and Results

The algorithm, as described above, serves to provide a baseline for comparison when evaluating the effects of additional changes to the algorithm.

Several areas of research were chosen to investigate further improvements and speedups (in both time and convergence speed). The areas of research targeted are: problem specific fitness functions, problem specific genetic operators, extinction events to introduce genetic diversity, and path correction heuristics. Each of these areas will be discussed in detail in the following sections.

Due to the iterative nature of the research, each set of changes will be presented in chronological order rather than grouped in another way. As such, some areas of focus,

like crossover, will appear multiple times, and some results may be updated when they are re-examined.

4.1 Crossover and Mass Extinction

4.1.1 Tested Components

Crossover

Let p_k be a member of the population represented by its chromosome, the waypoint sequence $[w_0, ..., w_{n_k}]$. p_k is partitioned into

$$p_{k[1]} = [w_0, ..., w_{i-1}]$$

and

$$p_{k[2]} = [w_i, ..., w_{n_k}]$$

for some cut point (index) $i \neq 0$. Neither sub-sequence can be empty. With this, given members p_1 and p_2, the algorithm chooses random cut-point at i in p_1 and at j in p_2 and creates two children c_1 and c_2 such that

$$c_1 = p_{1[1]} + p_{2[2]}$$

and

$$c_2 = p_{2[1]} + p_{1[2]}$$

After creation, the children may, with some probability, undergo mutation. As in the mutation-only version of the algorithm, a child population of size n is created and combined with the parent population of size n to create, temporarily, a population of size $2n$. The algorithm then resorts the combined population into domination fronts, and chooses the top n members to populate the next generation.

Mass Extinction

Though the concept of mass extinction exists in the genetic algorithms literature [20, 21], it has not been widely adopted in general and not at all for pathfinding. Therefore, the viability, or lack thereof, for mass extinction should be investigated for this problem. The intended goal of an extinction event is to jump start evolution when the process stalls, to get out of evolutionary dead-ends and avoid local optima.

Three versions of mass extinction are implemented, and experiments are run comparing each one with the base algorithm and each other. The first is entirely random while the second and third use a form of elitism to prevent eliminating the best genetic members of the population. In all three cases, the operator is invoked with probability p at an interval of 50 generations.

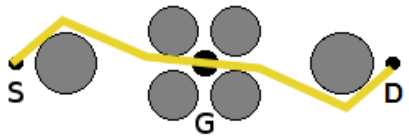
Random Regeneration. The first implementation of mass extinction is the simplest. 80% of the population is eliminated entirely at random. All eliminated members are replaced with randomly generated new members. While perhaps providing a reasonable baseline for comparison with other extinction implementations, this version may be problematic for vehicle flight. Completely random extinction events may eliminate all valid solutions within the population. Such an occurrence late in a run of the system would leave the vehicle without a valid path to fly. To avoid this, on-board the MAV, only those extinction implementations that incorporate elitism were used.

Regeneration via Crossover. In the second implementation of extinction, each extinction event eliminates 80% of the population. In a form of elitism, the best 3 members of the population are exempted from extinction. Re-population is a two-step process. First, 60% of the deleted members are replaced by randomly generated new members. The remaining 40% are created via crossover. Candidates for crossover are those members that survived extinction together with the newly created random members.

Regeneration via Mutation. Elitism plays a greater role in the third implementation. In this version, each extinction event eliminates all but the best 10 members of the population. Once again, re-population is a two step process. First, 10 randomly generated members are added to the population to ensure some degree of genetic diversity. Then, the balance of the population is created by randomly selecting one of the 10 members preserved through elitism, cloning it and mutating the clone.

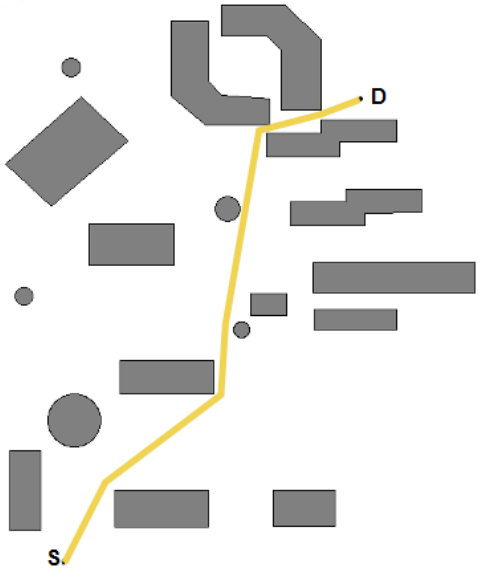
4.1.2 Experimental Method

The goal with these experiments is to determine the effects of crossover and mass extinction when used in a genetic algorithm for path-finding. To do this, the algorithm, and its various modifications, are run on a set of five sample inputs (Figure 4). Some maps, namely 1, 2, and 5, use intermediate goals that the path must intersect to change the difficulty of those inputs. In each case it makes the problem harder to solve by disallowing a straight line path from start to end. Each trial consisted of running the algorithm ten times for each input: with extinction probability p equal to 0.0, 0.05, 0.10, 0.15, 0.20, each with and without crossover enabled. Extinction occurs with probability p every 50 generations. For these tests, the population size was fixed at 100. For each mass extinction implementation, 40 trials are run for a total of 400 runs for each of the 5 inputs. In each trial, all runs for a given input use the same random seed, ensuring variance from the initial population can be neglected. For each input, a best solution is established by taking the best path after many runs of the control algorithm. The path length values from these solutions are used to establish two of the metrics used for experimental evaluation. In each trial, the mean number of generations is tracked for three measures: a valid solution, denoted *mng valid*; a solution with path length within 10% of the best found solution,*mng 10*; and a solution with path length within 5% of the best found solution, *mng 5*. Each run of the algorithm is limited to 3000 generations but will terminate prior to that if a 5% threshold solution is found. Runs that reach the maximum allowed 3000 generations are considered a failure, and the ratio of failed runs to successful runs is captured by the *success rate* calculated by $successful\_runs/total\_runs$.

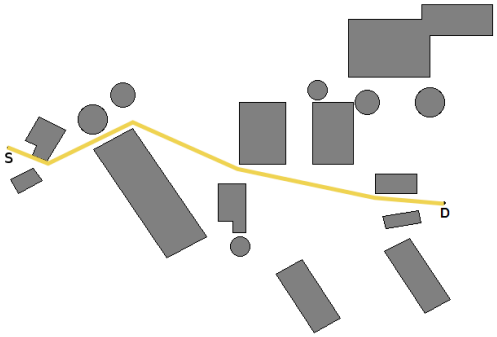


(a) Map 1 - Low Dificulty

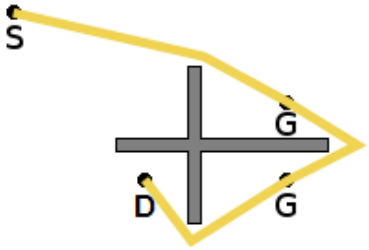
(b) Map 2 - Low Dificulty



(c) Map 3 - Medium Dificulty, Real World



(d) Map 4 - Low Dificulty, Real World



(e) Map 5 - Low Dificulty

Figure 4: Maps 1-5 used in the Mass Extinction and Crossover experiments. S indicates the start point, D indicates the destination, and G indicates an intermediate goal which must be visited. Yellow lines indicate a path within 5% of optimal.

4.1.3 Results

Crossover

The results show that the implementation of crossover had negative effects in four out of five input cases, with input 1 (Figure 4a) being the only instance for which the mean number of generations decreased when crossover was used. Additionally, the success rate, or the percent of all runs that found the optimal path, decreases significantly when crossover is enabled. In the case of input 4 (Figure 4d) the success rate drops as much as 32%. These effects can be seen by comparing the columns labeled "XO-0:EXT-0" and "XO-1:EXT-0" in Figures 5, 6, and 7. Note that Figures 5, 6, and 7 only show the data for reaching the *mgn 5* threshold described in the experimental design section. This data best classifies the algorithm's ability to solve the task, while the other data sets merely help establish the rate at which it is converging.

Mass Extinction

Mass extinction as a method of improving convergence speed does not appear to be holistically beneficial. Upon examining the mean number of generations, the first 5 column groups of Figures 5, 6, and 7 show either no discernible trend, or a positive trend, indicating an increase in computational effort. One notable exception is the effect of extinction 2 on input 3 (Figure 4c). Figure 6 shows a small negative trend, indicating improvement as the probability of events increases. Another notable effect is that of extinction 3 on input 1 (Figure 4a). When the probability was set to 10% or 15% there was a slight improvement over the control (Figure 7). Given the high degree of variance among the number of generations, as indicated by the black error bars showing ± 1 standard deviation, it is unlikely these effects are statistically significant. It is clear however that when examining the success rates there is in every case a positive trend when applying increasingly more extinction events. This combined with the results regarding mean generations, seems to suggest that the root cause of the algorithm's failure is lack of genetic diversity. As the probability of extinction increases, so to does the potential for new genetic material. It is reasonable to conclude that this increase would assist in finding solutions (as evidenced by the increase in success rates) but would make the search take longer on average (as evidenced in the rise in mean generations) because the algorithm must periodically waste time considering many extremely sub-optimal solutions.

Interactions

Seeing as each feature, crossover and mass extinction, is mostly negative independently, it is no surprise that when used in concert their negative effects are amplified. With crossover enabled, the mean number of generations dramatically rises as extinction probability grows. This trend can be seen in the last 5 column groups in Figures 5, 6, and 7. However with the exception of extinction 3 on input 3 (Figure 7 and 4c respectively) increasing values of extinction probability dramatically improve the success rate. The same conclusions discussed above hold here as well, extinction increases diversity, improving the capability of the search, but making it altogether slower.

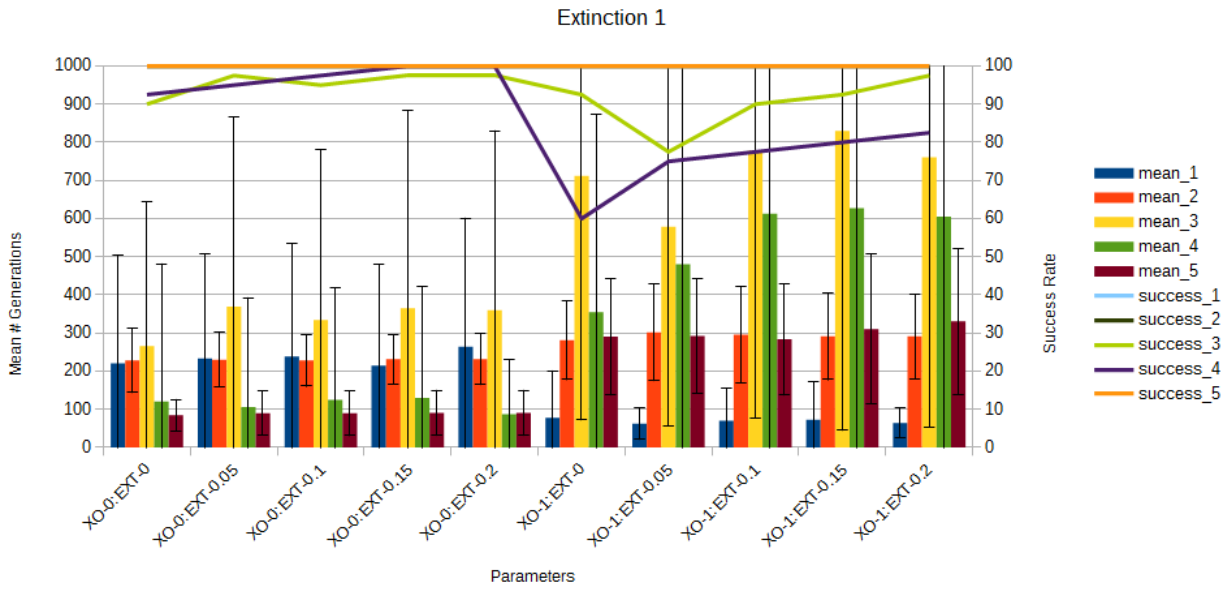


Figure 5: A graph showing the mean number of generations for successful runs of each input (bars) and the success rate of each set of parameters (lines) for extinction method 1. ± one standard deviation is shown as black error bars. Inputs tested were 1-5.

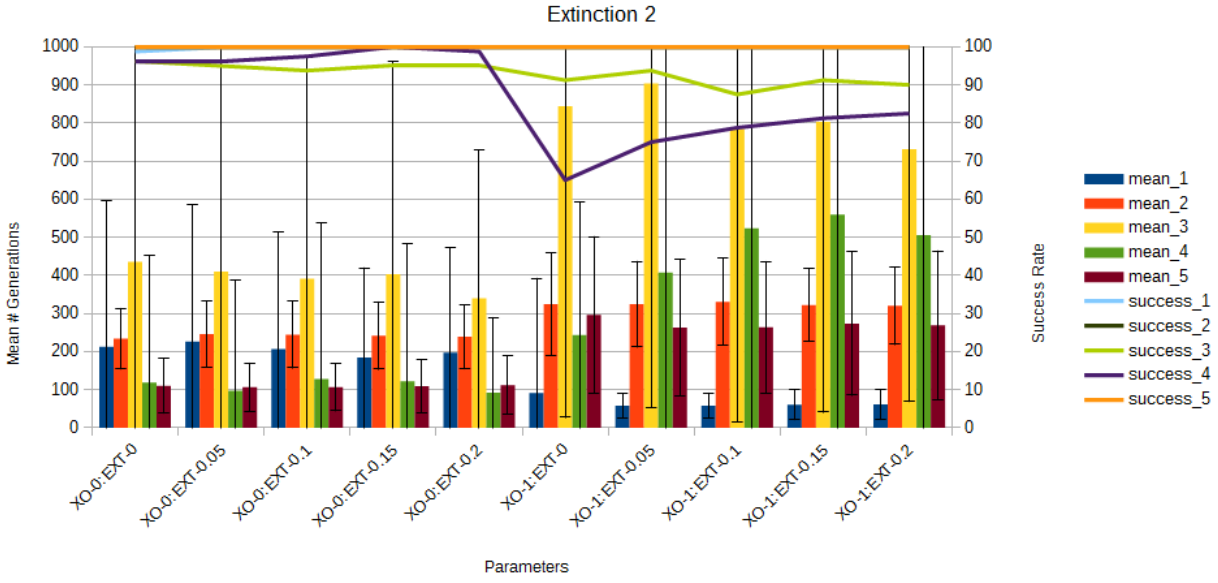


Figure 6: A graph showing the mean number of generations for successful runs of each input (bars) and the success rate of each set of parameters (lines) for extinction method 2. ± one standard deviation is shown as black error bars. Inputs tested were 1-5.

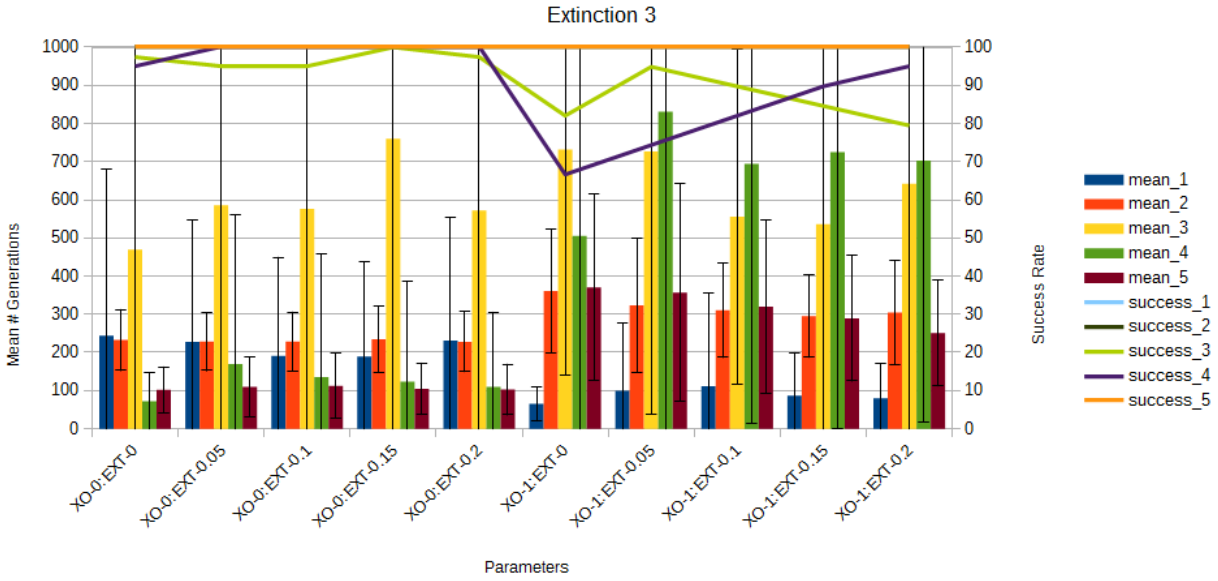


Figure 7: A graph showing the mean number of generations for successful runs of each input (bars) and the success rate of each set of parameters (lines) for extinction method 3. ± one standard deviation is shown as black error bars. Inputs tested were 1-5.

4.2 Crossover, Obstacle Intrusion, Path Correction, and Mutation Size

4.2.1 Tested Components

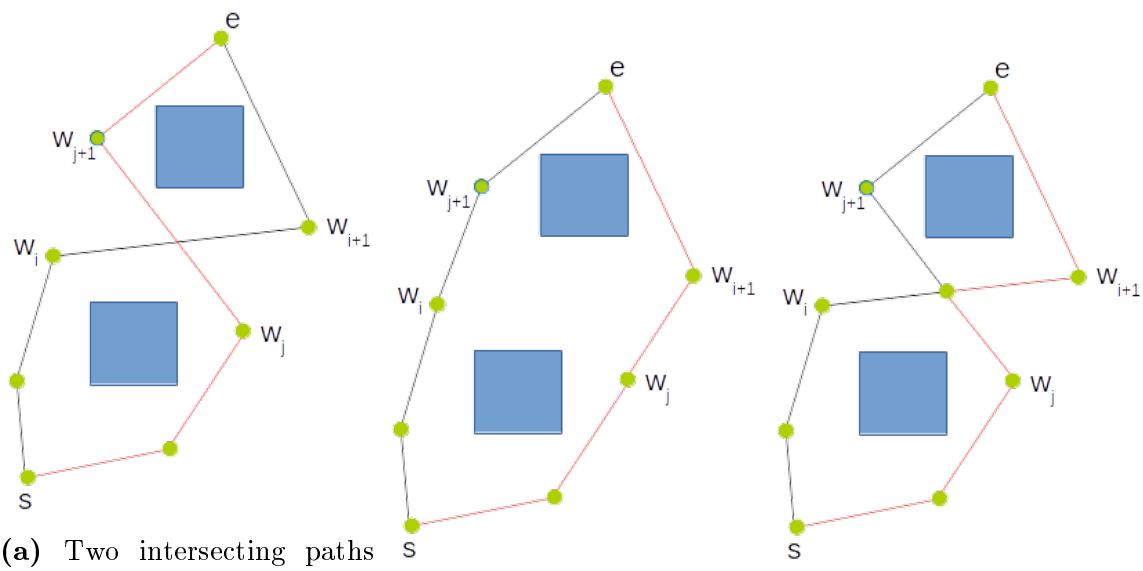
Intersection Crossover

Though typical for genetic path planning algorithms, the single-point crossover described in Section 1.4 is largely ineffective [22]. This is, perhaps, not surprising given that the genome represents spatial information but there is no spatial basis for choosing cut locations or applying crossover. To address this, a new form of crossover specific to path planning is introduced, called *intersection crossover*. This operator allows crossover only between genomes representing paths that intersect within the environment.

In intersection crossover, parent selection occurs via tournament selection as for XO-R. Once parents have been chosen, crossover occurs only if the paths they encode intersect. Otherwise, rather than attempt multiple times to choose intersecting paths, each parent is copied and the copies mutated.

Two versions of intersection crossover are implemented: (1) XO-I and (2) XO-I$^+$. To define them formally, let p_1 and p_2 be members of the population with genomes $\langle s, w_{0_1}, \ldots, w_i, w_{i+1}, \ldots, w_{\ell_1}, e\rangle$ and $\langle s, w_{0_2}, \ldots, w_j, w_{j+1}, \ldots, w_{\ell_2}, e\rangle$, respectively. Further, let p_1 and p_2 intersect at point y on path segments $\langle w_i, w_{i+1}\rangle$ and $\langle w_j, w_{j+1}\rangle$.

Figure 8 illustrates XO-I and XO-I$^+$. In XO-I, child $c_1 = \langle s, w_{0_1}, \ldots, w_i, w_{j+1}, \ldots, w_{\ell_2}, e\rangle$ and child $c_2 = \langle s, w_{0_2}, \ldots, w_j, w_{i+1}, \ldots, w_{\ell_1}, e\rangle$. Informally, c_1 includes the waypoints of p_1 from s up to w_i, the waypoint immediately before intersection point y, and the waypoints of p_2 from w_{j+1}, the waypoint immediately after y, to e.



(a) Two intersecting paths chosen as parents for XO-I or XO-I$^+$.

(b) The two paths from Figure 8a after XO-I.

(c) The two paths from Figure 8a after XO-I$^+$.

Figure 8: Depiction of intersection crossover. The first image shows two intersecting paths. The middle and right show the result using XO-I and XO-I$^+$, respectively.

XO-I$^+$ augments each child with intersection point y. This yields children $c_1 = \langle s, w_{0_1}, \ldots, w_i, y, w_{j+1}, \ldots, w_{\ell_2}, e\rangle$ and $c_2 = \langle s, w_{0_2}, \ldots, w_j, y, w_{i+1}, \ldots, w_{\ell_1}, e\rangle$.

Obstacle Intrusion

The standard objective related to obstacle avoidance is a simple count of the number of obstacles hit. Minimizing this quantity to 0 yields a collision-free path; however, this measure of collisions has a shortcoming: a path that crashes through the middle of an obstacle is indistinguishable (with respect to that objective) from one that only slightly clips a corner of an obstacle. This limits the evolutionary utility of the obstacle avoidance objective.

An alternative real-valued measure of collision, called *obstacle intrusion*, is presented based on the degree to which a path intersects an obstacle. The resulting finer fitness granularity improves the algorithm's ability to distinguish and reward degrees of obstacle intrusion, thus facilitating evolutionary search.

The *obstacle intrusion* value is calculated as follows. The points at which the path enters and exits an obstacle $o_j \in O$ are determined. As there can be multiple enter/exit pairs for o_j, the i-th pair is denoted as $pair_i = (o_{j:in[i]}, o_{j:out[i]})$. Connected by the path segments between them, each pair divides the obstacle into two regions with areas A_1^i and A_2^i. The obstacle intrusion value for $pair_i$ on obstacle o_j is $I_j^i = min(A_1^i, A_2^i)$. The total obstacle intrusion value for o_j is $I_j = \sum_i I_j^i$ and the total intrusion value for a population member as $\sum_{j \in |O|} I_j$, the sum over all obstacles. See Figure 9.

It would likely improve evolution to have a separate objective for each obstacle in the environment. However, this introduces implementation problems as the number of objectives would be both variable and large. While the objective as presented here is a compromise in

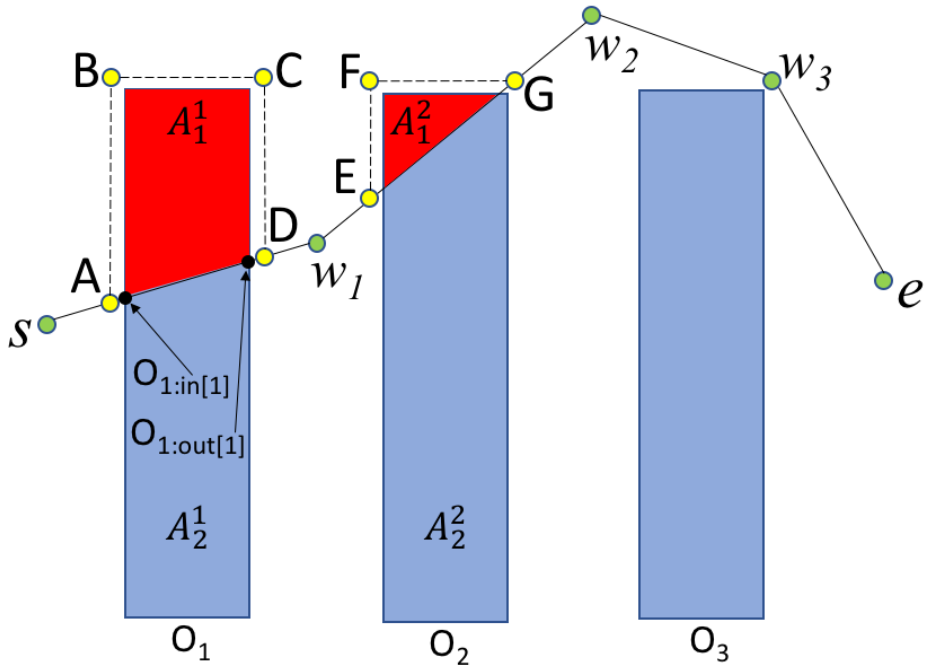


Figure 9: The obstacle intrusion measure and path correction operator use the same information in different ways. Consider path $p = \langle s, w_1, w_2, w_3, e\rangle$ through this simple map. The value of the obstacle intrusion objective for this input/path pair is $A_1^1 + A_1^2$. Applying path correction to p results in path $p' = \langle s, A, B, C, D, w_1, E, F, G, w_2, w_3, e\rangle$.

this sense, it proves extremely beneficial in testing.

Path Correction

The *path correction* operator eliminates obstacle collisions by "pulling" a path off of any obstacles it hits. Consider again the points at which a path enters and exits an obstacle. The algorithm determines the shortest path, along the obstacle perimeter, between these two points. It then replaces in the genome the path segments between entry and exit with a new subpath following the shortest path along the perimeter, translated 1m outside the perimeter. Figure 9 shows the effect of path correction.

If used, path correction can be applied once, to the initial population, or repeatedy at some fixed interval. When path correction is applied to the initial population, it is denoted by IPC. When it is applied at a fixed interval of x generations, it is denoted by RPC-x. RPC-0 denotes the case in which path correction is not applied repeatedly. The hypothesis is that applying path correction too frequently might inhibit evolution by limiting exploitation.

4.2.2 Experimental Method

The experiments described here are designed to evaluate the effectiveness of the new operators and optimization objective introduced. Each experiment consists of 80 trials where each trial includes runs with each of the combinations of parameters being tested. Trials are run for all four maps (Figures 10a-10d). Within a trial, the initial population is the

Feature	Values Tested	Explanation	Other Parameters
obs_intrusion	*off, on*		*RPC-0, XO-I*$^+$
path_correction	*RPC-0* *IPC* *RPC-100* *IPC* $\wedge$ *RPC-100*	*no path correction* *path correct initial population* *path correct every 100 generations* *path correct initial pop & every 100 gens*	*obs_intrusion*
correction_interval	*RPC-{1, 5, 20, 50, 100}*	*path correction every X generations*	*obs_intrusion, XO-I*$^+$*, [IPC: off, on]*
crossover	*XO-Off* *XO-R* *XO-I* *XO-I*$^+$	*no crossover* *random cutpoint crossover* *intersection crossover* *intersection crossover, point added*	*obs_intrusion, [RPC-0, RPC-20]*
avg_move	*8, 12, 16, 24, 32, 40, 48, 64*	*average distance of move mutation*	*obs_intrusion, RPC-20,* *[XO-Off, XO-R, XO-I, XO-I*$^+$*]*

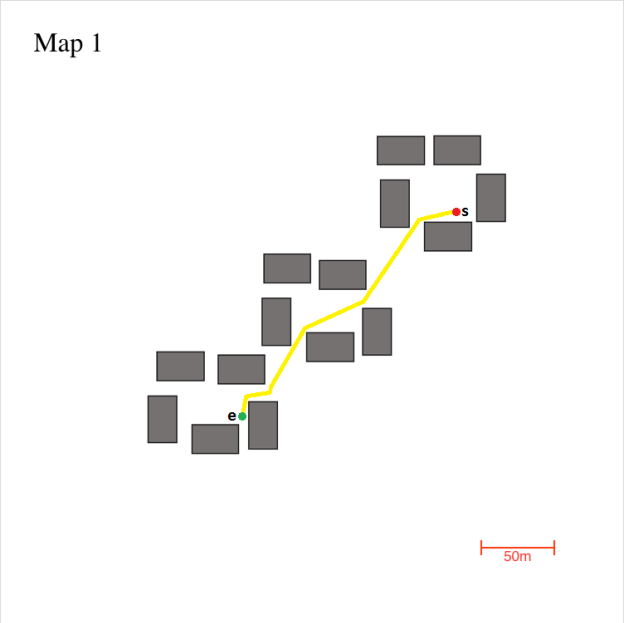
Table 1: Overview of the experiments reported. The first and second columns list the feature that is the primary subject of the experiment and the values being tested. The 4th column provides values for other relevant parameters. As indicated, some experiments used all new features in concert.

same for each run to eliminate a source of variability. Experiments are run on the Stampede supercomputer at the Texas Advanced Computing Center. Each standard compute node on Stampede includes two Intel Xeon E5-2680 8 core Sandy Bridge processors and 32GB of RAM.

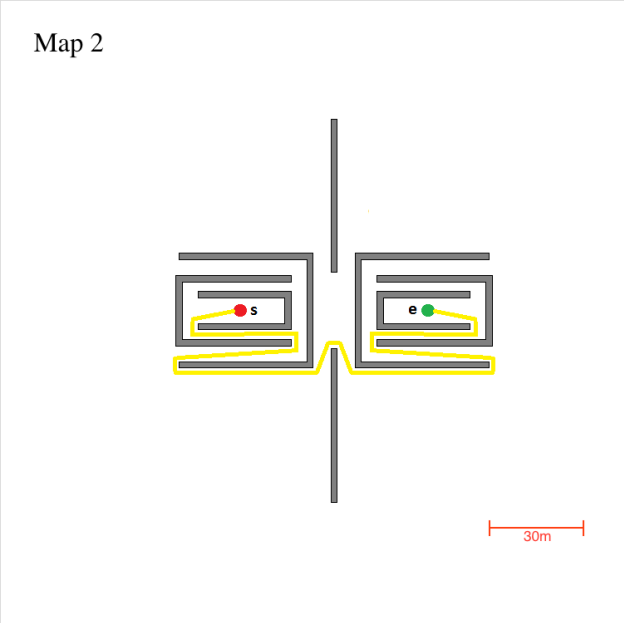
The values of several parameters used in this work were established empirically in prior works [23, 22]. These include the probabilities with which the four mutation operators are applied (0.25, 0.15, 0.1, 0.5, respectively), and the post-crossover mutation probability (0.0). In addition, the utility of the waypoint count objective is found to be somewhat beneficial. Other parameter values are established empirically as part of this work, including the crossover probability (0.4) and whether path correction is applied to generation 0 (no).

As discussed in the problem statement, successful path planning requires satisfying the validity conditions as well as minimizing, to the extent possible, the optimization objectives. Given the desire to improve the efficiency of path planning, the number of generations required to find a valid path of acceptable length is also examined. As unsuccessful runs are expensive, the success rates of the trials are measured. In this context, success is defined as finding a path within 5% of optimal in fewer than a specified maximum number of generations. For these experiments the maximum generations is 3000. Within the set of successful runs, the mean number of generations and the standard deviations are tracked.

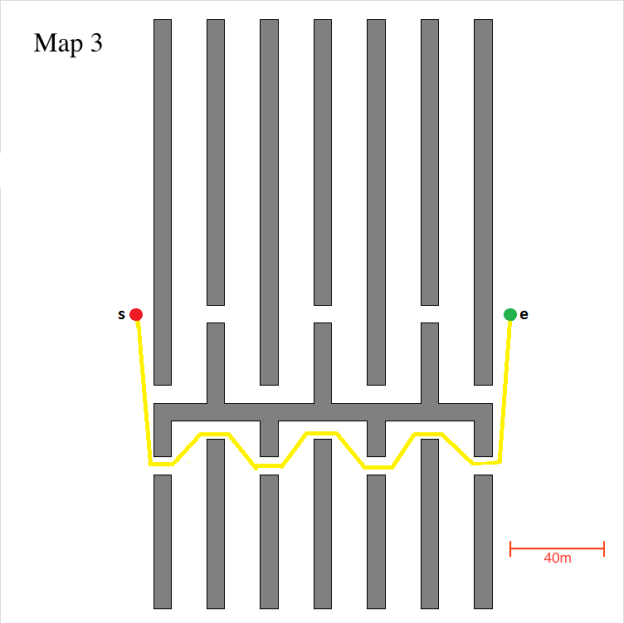
Table 1 summarizes the sequence of experiments. The experiments are conducted in sequence so that once the utility of a feature is established, it can be used, or excluded, to enhance overall performance in subsequent experiments. Thus, experiments listed lower in Table 1 include most or all of the features described.



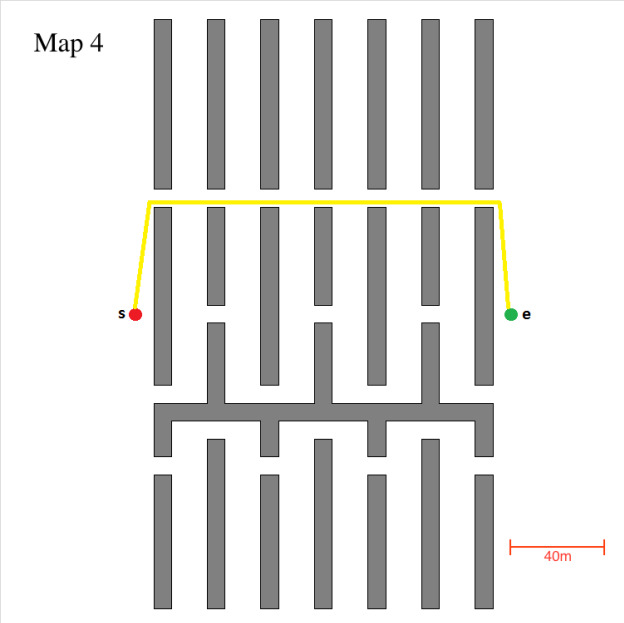
(a) Map 6 - Medium Difficulty, Real World



(b) Map 7 - Hard Difficulty



(c) Map 8 - Hard Difficulty



(d) Map 9 - Hard Difficulty

Figure 10: Maps 6-9 used in the Intersection Crossover, Path Correction, Obstacle Intrusion, and Mutation Size experiments. S indicates the start point, E indicates the end point, and the yellow line indicates a path within 5% of optimal.

4.2.3 Results

For each map tested, an optimal path length is established. A run is considered successful if it finds a valid path with a length within 5% of optimal. The success rate is the percentage of runs that are successful. For each experiment, the success rate (S-R), the mean number

	XO-Off		XO-R		XO-I		XO-I$^+$	
	RPC-0	**RPC-20**	**RPC-0**	**RPC-20**	**RPC-0**	**RPC-20**	**RPC-0**	**RPC-20**
	Map 1							
S-R	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
S-M	503.60	38.09	337.85	35.25	259.28	31.96	163.66	32.18
S-SD	255.55	7.98	193.14	5.81	128.67	6.85	75.29	7.09
	Map 2							
S-R	0.00%	99.87%	0.00%	100.00%	0.00%	100.00%	59.00%	100.00%
S-M	N/A	79.75	N/A	71.42	N/A	43.48	1501.58	46.90
S-SD	N/A	65.00	N/A	17.20	N/A	9.39	611.63	15.31
	Map 3							
S-R	4.26%	85.16%	1.06%	82.55%	52.13%	87.89%	69.15%	80.99%
S-M	2250.25	184.03	2898.00	259.66	2126.94	136.69	734.38	160.24
S-SD	491.19	282.42	0.00	474.73	528.17	282.65	280.95	345.63
	Map 4							
S-R	71.88%	90.89%	80.21%	82.55%	82.29%	86.98%	69.79%	85.81%
S-M	73.20	65.32	287.22	259.66	86.63	44.79	130.31	58.83
S-SD	34.08	77.15	581.00	474.73	228.90	45.55	374.46	94.83
	Across Four Maps							
S-R	44.03%	93.98%	45.32%	92.32%	58.61%	93.72%	74.49%	91.70%
S-M	942.35	91.79	1174.36	104.97	824.28	64.23	632.48	74.54
S-SD	260.27	108.13	258.05	143.30	295.24	86.11	335.58	115.72

Table 2: Crossover comparisons across four maps.
Comparison of no path-correction (RPC-0) versus path correction every 20th generation, without validating the initial population (RPC-20), across mutation distances centered at 8, 12, 16, 24, 32, 40, 48, and 64 meters. S-R, S-M, and S-SD represent success rate and the mean and standard deviation, in number of generations, for successful runs.

of generations required for success (S-M), and the standard deviation in the number of generations (S-SD) are reported.

Intersection Crossover

Table 2 compares the effects of having no crossover versus standard random cut-point crossover (XO-R) versus two new crossovers that employ path intersections as the basis for defining cut points (XO-I and XO-I$^+$).

When looking at averages across all maps, XO-R leads to a comparable success rate as having no crossover, while increasing both the number of generations to find a solution as well as the standard deviation. These findings, along with the additional cost of performing crossover, suggest that no crossover is preferred to XO-R.

When comparing no crossover to XO-I and XO-I$^+$, the new crossovers have a definite advantage. Both new operators have comparable success rates, coupled with a decreased number of generations and standard deviation. On a map by map basis, the advantages are not obvious on maps 1 and 2, but become more apparent on maps 3 and 4, although no clear winner emerges. The superiority of the new crossovers stems from their leveraging of the existing intersection points, which indicate the crossing paths are spatially compatible and could exchange segments before and after the intersection point. If paths do not already intersect, swapping random segments is likely to be strongly disruptive to the formed

	Success Rate (S-R)		Success Mean (S-M)		Success SD (S-SD)	
	Off	**On**	**Off**	**On**	**Off**	**On**
Map 1	37.50%	100.00%	903.86	162.23	870.03	76.62
Map 2	0.00%	67.42%	N/A	1473.57	N/A	678.06
Map 3	0.00%	82.11%	N/A	831.99	N/A	388.29
Map 4	1.09%	77.17%	12.00	71.21	N/A	41.75
Average	9.65%	81.67%	457.93	634.75	870.03	296.19

Table 3: Effect of obstacle intrusion on all maps.
Comparison of runs on maps 1 through 4 with obstacle intrusion off and on. Runs were performed without path correction and with XO-I$^+$. S-M and S-SD are presented as number of generations.

solutions, complicating exploitation of evolved sub-paths.

Obstacle Intrusion

To determine the impact of obstacle intrusion, trials that isolate it from the path correction operator are run. For the maps tested, strong evidence is presented which shows that all aspects of the algorithm's performance benefit from employing this new objective. Specifically, success rate increases dramatically, mean generations decrease, standard deviations of generations for successful runs decrease, and the percentage of successful runs drastically increases.

In particular, only the least complex map, map 1, has significant success (37.5%) without obstacle intrusion. Maps 2, 3, and 4 have success rates of $0\%, 0\%$, and 1.09%, respectively. The 1.09% value for map 4 represents a single successful run. With obstacle intrusion enabled, the success rates jump to $100\%, 67.42\%, 82.10\%$, and 77.17%, respectively.

When examining the mean number of generations, there is a seemingly negative effect of obstacle intrusion on map 4. This is due to an outlying event where a single success is achieved extremely fast while obstacle intrusion is off. Thus, the mean generation measure is artificially low for the obstacle intrusion case. This behavior, however, is not representative of having obstacle intrusion turned off for said map, as evidenced by the 1% success rate, as opposed to the 77% success rate obtained when obstacle intrusion is on. Table 3 summarizes the results.

The cost of obstacle intrusion is further analyzed in terms of actual run-times. The goal is to ensure that a decrease in the number of mean generations is not offset by a commensurate increase in the time to evolve each generation. With obstacle intrusion on, each generation can take up to 3 times as long as generations with obstacle intrusion off. However, since the success rates increase dramatically and the overall number of generations decreases, the additional time per generation is warranted. Only map 1 has sufficiently high success rates in both cases to allow meaningful comparison. Average time for successful runs with obstacle intrusion off is 170.85 versus 50.32 seconds when the objective is on.

Path Correction

Figure 11 summarizes the results of the path correction tests. The graph shows that, on the less deceptive maps (namely, maps 1 and 2), most path-correction interval setups succeed

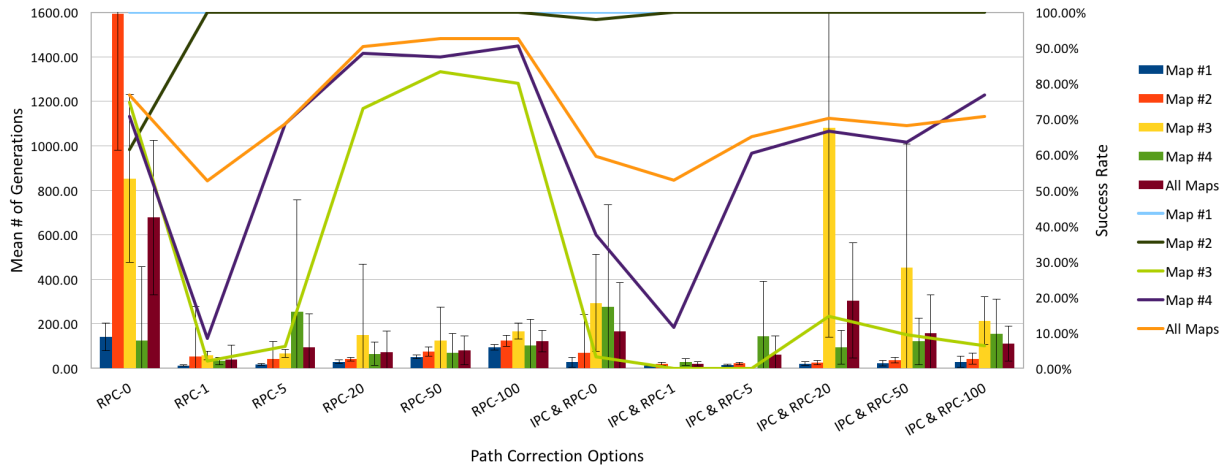


Figure 11: Results for experiments on the frequency with which path correction is applied. IPC indicates that members of the initial population were path corrected. Obstacle intrusion was on in all cases. The lines and scale on the right indicate success rates for each of the four maps.

in all cases (light blue and black lines) with the exceptions of "never" (RPC-0) and "only on the initial generation"(IPC & RPC-0). Nevertheless, the number of generations required to find a successful path does vary with parameters. On these maps, path-correcting more often tends to lead to improved results; however, the differences are modest.

On the more deceptive maps (*i.e.* maps 3 and 4), path-correction setups have a stronger distinguishing effect (note the differences among the yellow and green bars across the path-correction intervals). Specifically, not validating the initial population appears to allow for increased initial exploration, an effect that is strengthened when the path-correcting interval is larger, thus increasing the initial exploration period (note the generally lower bars on the left-most set of RPC-1 – RPC-100 intervals, versus the taller bars on the right-most set of IPC & RPC-1 – RPC-100 in figure 11). Based on these findings, it appears that waiting to path-correct the population is likely to lead to better exploration of the solution space. In the experiments, best overall performances, as measured by a combination of high success rate and low average number of generations, are obtained by the setups of path-correcting every 20 or every 50 generations, coupled with NOT path-correcting the initial population.

Additionally, when comparing path-correction intervals against runs with no path-correction (RPC-0), the difference can be stark but is map specific. For example, while only a small difference is observed on map 1 (compare first dark blue bar to the others), map 2 strongly benefits from path-correcting at any interval (compare the first red bar on the graph to the others). On maps 3 and 4, however, the benefit of path-correction isn't as clear. In fact, many setups lead to severely decreased success rates. Nevertheless, among the beneficial setups an increase in the number of successful runs is observed, as well as a decrease in the number of generations and standard deviations.

While future testing will include a variety of larger maps, real world scenarios are generally likely to be less deceptive than the scenarios tested in this work (closer to map 1, for instance). Nevertheless, since the specifics of the deceptiveness may vary, future algorithm improvements would likely revolve around increasing its adaptability to the unpredictable

nature of the map. Since it is generally advisable to conduct multiple evolution runs, one sensible option is to employ various path-correction intervals across these runs to mediate the impossibility of knowing the specifics of the sought after path before it is found by the GA.

Mutation Size

To further verify the observed crossover differences, a number of different mutation sizes are tested (mutation distances centered at 8, 12, 16, 24, 32, 40, 48, and 64 meters). When looking at setups with each of the different mutation sizes for each of the four crossover types (no crossover, XO-R, XO-I, and XO-I$^+$), no clear winning distance is seen for all maps. While it is to be expected that the ideal average path mutation distance is map specific, a general trend was observed: evolution behavior generally improves as mutation distance increases, approaching some map-specific value, and decreases again as this value is surpassed.

As before, on maps 1 and 2 no significant differences are observed upon increasing mutation distances. On map 4, higher mutation distances appear to be beneficial. It is believed that this is a result of a straight segment being part of the sought after, but more difficult to find, ideal path. On Map 3, increasing average mutation size beyond 40 m is no longer beneficial, which can be explained by the map's lack of a straight path as found in map 4.

Based on the findings, an average mutation of 24-32 meters appears to work well for the tested scenarios. For the obstacle spacing and shapes present, these distances resulted in an effective compromise between exploring the space and exploiting solution features. Given that beneficial mutation distances are map specific, it would be sensible to incorporate dynamically adjusting mutation distances. Note that since no winning mutation size emerged, crossover comparison values presented in Table 2 are averaged across all of the tested mutation distances (8 through 64 meters).

5 Final Conclusions

5.1 Crossover and Mass Extinction

In this set of experiments, the effects of crossover and mass extinction on a genetic algorithm are tested for pathfinding in a continuous environment. Due to the large number of possible algorithms and parameters as well as variations in environments, a definitive statement regarding the utility of these operators can not be made. Broader experimentation is required to explore these variables. However, the results show that crossover may not be desirable for some instances of pathfinding while mass extinction shows some promise. The inputs that were used for testing are varied. Numbers 1 and 2 are linear and symmetric yet the results for these inputs are very different. Crossover helps with 1 but not 2 and extinction is slightly negative for 1 and neutral for 2. Inputs 3 and 4 are drawn from the real world and are significantly more complex. In both cases, crossover is a negative influence. Input 5 is simple but not symmetric. In this case, crossover is negative while extinction is neutral.

When extinction exhibits a benefit, it is often best for *mng_5*, the threshold that requires the largest number of generations to reach. This makes intuitive sense: because extinction occurs infrequently, longer times provide greater opportunity for extinction to benefit the

population. In fact, for *mng_5*, benefit due to extinction increases slightly with extinction probability. The reverse is true for *mng_valid*.

In the experiments, crossover and extinction coexist reasonably well. For some inputs, extinction improves results when used with crossover while in others extinction is neutral. In no case is the combination significantly negative for measure *mng_5*, though for most inputs crossover is negative. However, this effect is independent of extinction.

There is a great deal of future work to continue this research. As mentioned above, greater variation in extinction implementations and parameters, such as probability and the frequency with which extinction is invoked, could expose trends. For example, rather than invoking extinction at regular intervals, it could be used based on a decreasing rate of change in the population. Alternatively, the increasing utility of extinction with number of generations might suggest increasing the extinction probability with time.

Another area for future work is the extent to which random regeneration aids extinction. Extinction implementations 2 and 3 vary significantly in the amount of random regeneration. It is hypothesized that the new genetic material introduced by random members could benefit evolution. Experiments in which both versions are used with varying numbers of random new members might identify a trend and the ideal random percentage.

5.2 Crossover, Obstacle Intrusion, and Path Correction

In this set of experiments two new domain-specific genetic operators and a new domain-specific optimization objective were presented that each dramatically improve the success rate of the algorithm and the number of generations required to achieve success. Further, the maps for which the algorithm can successfully find paths are significantly more complex and difficult to solve than in previous iterations of pathfinder.

In future work, experiments should be conducted to determine if using XO-I and XO-I+ in concert proves beneficial. This is motivated by the data showing that for some maps XO-I is significantly better than XO-I+ while for others the situation is reversed. Perhaps it is possible to find a way to leverage the benefits of both within a single run.

Some parameters, such as the move mutation distance and the path correction frequency, demonstrate utility for all maps but with different values. For example, the algorithm performs better with path correction every 50 generations for map 3 but every 20 generations for other maps. It might be possible to leverage features in the environment or changes in evolutionary progress to change these parameter values dynamically.

In another direction, it might be useful to explore improvements to obstacle intrusion. For example, perhaps it is possible to maintain this objective value separately for each obstacle without creating an unmanageably large number of objectives. Further, the current implementation of obstacle intrusion is complex. It may be possible to simplify it and, in the process, make it more efficient.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms.* The MIT Press, 2014.

[2] K. A. De Jong, *Evolutionary Computation A Unified Approach.* Bradford Books, 2016.

[3] V. Pareto, "The new theories of economics," *Journal of Political Economy*, vol. 5, no. 4, p. 485–502, 1897.

[4] N. Srinivas and K. Deb, "Multiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, no. 3, pp. 221–248, 1994.

[5] O. Goldreich, *Computational complexity: a conceptual perspective.* Cambridge University Press, 2008.

[6] M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[7] O. Goldreich, *P, NP, and NP-Completeness.* Cambridge University Press, 2010.

[8] R. A. Bailey, P. J. Cameron, and R. Connelly, "Sudoku, gerechte designs, resolutions, affine space, spreads, reguli, and hamming codes," *The American Mathematical Monthly*, vol. 115, p. 383–404, May 2008.

[9] Y. K. Hwang and N. Ahuja, "Gross motion planning – a survey," *ACM Computing Surveys*, vol. 24, no. 3, pp. 219–291, 1992.

[10] F. Ahmed and K. Deb, "Multi-objective optimal path planning using elitist non-dominated sorting genetic algorithms," Tech. Rep. 20111013, Kanpur Genetic Algorithms Laboratory (KanGAL), Indian Institute of Technology, 2011.

[11] I. Hasircioglu, H. Topcuoglu, and M. Ermis, "3-d path planning for the navigation of unmanned aerial vehicles by using evolutionary algorithms," in *Genetic Algorithms and Evolutionary Computation Conference*, pp. 1499–1506, ACM, 2008.

[12] H. Jun and Z. Qingbao, "Multi-objective mobile robot path planning based on improved genetic algorithm," in *IEEE International Conference on Intelligent Computation Technology and Automation*, pp. 752–756, 2010.

[13] J. Xiao, Z. Michalewicz, L. Zhang, and K. Trojanowski, "Adaptive evolutionary planner/navigator for mobile robots," *IEEE Trans Evol Comp*, vol. 1, no. 1, 1997.

[14] C. Zheng, M. Ding, C. Zhou, , and L. Li, "Coevolving and cooperating path planner for multiple unmanned air vehicles," *Engineering Applications of Artificial Intelligence*, vol. 17, pp. 887–896, 2004.

[15] J. H. Holland, "Outline for a logical theory of adaptive systems," *Journal of the ACM*, vol. 9, p. 297–314, Jan 1962.

[16] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Trans Evol. Comp.*, vol. 6, no. 2, pp. 182–197, 2002.

[17] A. Hermanu, T. Manikas, K. Ashenayi, and R. Wainwright, "Autonomous robot navigation using a genetic algorithm with an efficient genotype structure," in *Intelligent Engineering Systems Through Artificial Neural Networks: Smart Engineering Systems Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems and Artificial Life*, ASME Press, 2004.

[18] K. Sedighi, K. Ashenayi, T. Manikas, R. Wainwright, and H.-M. Tai, "Autonomous local path planning for a mobile robot using a genetic algorithm," in *Proceedings of Congress on Evolutionary Computation*, pp. 1338–1345, IEEE, 2004.

[19] O. Mengshoel and D. Goldberg, "The crowding approach to niching in genetic algorithms," *Evolutionary Computation*, vol. 16, no. 3, pp. 315–354, 2008.

[20] B. Jaworski, L. Kuczkowski, R. Smierzchalski, and P. Kolendo, "Extinction event concepts for the evolutionary algorithms," *Przeglad Elektrotechniczny (Electrical Review)*, vol. 88, no. 10b, 2012.

[21] J. Lehman and R. Miikkulainen, "Extinction events can accelerate evolution," *PLoS ONE*, vol. 10, August 2015.

[22] D. Mathias and V. Ragusa, "An empirical study of crossover and mass extinction in a genetic algorithm for pathfinding in a continuous environment," in *Proceedings of Congress on Evolutionary Computation*, IEEE, 2016.

[23] D. Mathias and V. Ragusa, "Micro aerial vehicle pathfinding and flight using a multiobjective genetic algorithm," in *Proc. Intelligent Systems Conf.*, 2016.