Memorandum

TO: Shawki M. Areibi Ph.D., P.Eng

FROM: Basil Debowski, Jo VandenDool, & Kyle Binkley

SUBJECT: Final Report
DATE: Nov. 30, 2008

Over the past 3 months Path Planners has worked to improve and enhance the dynamic path planning system that was provided to us in September. Three major areas of interest were considered and developed to determine their affects on the complex system.

The first component was a particle swarm optimization algorithm to replace the existing genetic or evolutionary algorithm. It was believed that this would improve the efficiency of convergence to an optimal solution. Unfortunately this was not the case. The second component addressed collision avoidance and attempted to scale path fitness according to 3 factors: type of obstacle collision occurs with, time to collision and number of collisions. The last component was a radial vision system which is designed to cause the robot to respond more to its local environment than its global environment.

The following report outlines in detail the decisions that were made in the planning phase, the design methodology that was used to carry out these decisions, and analysis of the results that were obtained through the testing of the implemented components.

Robot Dynamic Environment Path Planning

Final Report

Basil Debowski Jo VandenDool Kyle Binkley

Sunday, November 30, 2008

University of Guelph

Executive Summary

This is the final report documenting the work of Path Planners on the dynamic path planning project proposed in September. The intention of this report is to cover all aspects of the additions that were made to the system from the planning phase to the analysis of the results that were obtained through testing of implemented work.

Two of the tasks approach the fitness function currently used in the path planning system and attempt to improve collision avoidance and path feasibility. One of the tasks works by projecting the path of obstacles in the direction of movement and assessing the interactions of the projected paths with the planned path of the robot. The other assesses how imminent a collision is by considering the distance of the collision from the robots current location.

The third task will be to implement a particle swarm optimization (PSO) algorithm with which to compare to the current genetic algorithm with respect to algorithm performance as well as computational complexity. It is yet to be determined if dynamic PSO parameters will be implemented.

Currently these three tasks are simultaneously under development and testing is scheduled to commence early November.

Nomenclature

In the following report we will be using the following abbreviations and terminology.

Efficiency - Path Efficiency refers to the length of the path as well as smoothness in the path. Short and straight paths are more efficient than long paths with sharp turns.

Collision - A section of a path that intersects with an obstacle, or the event of the robot touching an obstacle.

Collision Avoidance - A system that works to prevent the robot from colliding with an obstacle.

Cost Evaluation Function - This function acts as an entry point to a number of functions designed to alter the fitness of a path after the fitness is calculated by other functions in the system.

***Cost* -** A number that reflects the distance from optimality of the path. Zero cost is best. Cost is the inverse of Fitness.

Feasible Path - Any path from the Robot’s current location to the target that does not intersect with an obstacle.

Feasible Node - Any node which lays within the map and outside of all obstacles.

***Fitness* -** A calculated number that reflects the nearness to optimality of the path. Fitness is the inverse of cost.

GA: Genetic Algorithm **-** A meta heuristic algorithm that works by combining solutions of known quality to generate new solutions in an attempt to improve quality.

GAP: Genetic Algorithm Planner **-** The path planning algorithm originally implemented in the project. Paths were planned by a GA.

Local Minimum - An area of a solution map that is better than surrounding area but not optimal.

Local Search **-** An algorithm designed to find the best nearby solution within a solution space.

Operator, Smooth **-** A function that inserts nodes into a path to reduce the radius of turning in corners.

Operator, Repair **-** A function that attempts to make infeasible paths feasible by adding nodes to paths in areas of obstacle intersection.

Operator, Mutation - A function that randomly alters paths to increase genetic diversity.

Optimal Path - A theoretical best possible path. GA & PSO attempt to converge on this path.

PSO: Particle Swarm Optimization **-** A meta heuristic algorithm that works by assigning solutions positions and velocities and moving them in an effort to find better quality solutions.

PSAP: Particle Swarm Algorithm Planner **-** The path planning algorithm using PSO.

Radial Vision **-** A system that constrains the distance down a path that the robot uses to analyze which path is best to follow.

Safety **-** Path Safety refers to the likeliness of a collision if the robot follows a particular path; the less likely a collision, the safer the path.

Smoothness - Path smoothness refers to the radius of turning in a section of a path; greater radii are smoother.

Solution Space - The set of all feasible solutions to a problem.

Table of Contents

1 Introduction

1.1 Problem Statement

It is not unreasonable to believe that in not too distant future, ideas like mobile robots and self-driven cars will be an everyday occurrence. These ideas both rely on the concept of mobile robot path planning in a dynamic environment. This concept deals with planning a path of motion for a mobile robot or vehicle, through an environment that consists of static and moving obstacles. The path that is planned must be short, safe, low-power for the robot to follow, and free of collisions. This type of problem is called a multi-objective problem since it deals with optimizing multiple objectives (length, safety, smoothness). Several techniques have been investigated in solving this problem and some solutions do exist. The research and existing solutions are in there early stages and much work can be done to improve existing ideas.

The project completed by Path Planners for The University of Guelph is an improvement of an existing project. The project expanded on an existing computer simulation which guides a simulated robot through a two-dimensional environment among static and moving obstacles towards a target destination. The target destination is also able to move through the environment. The previously implemented solution used a GAP (Genetic Algorithm Planner) to find optimal paths through the dynamic environment. The GAP was originally designed to plan paths for a non-moving robot through a dynamic environment in which objects were static in motion but could be added and removed from the map, towards a static target. The GAP worked well in finding optimal solutions in this environment. Also, a Local Search algorithm was developed to possibly aid the GAP, but it was never correctly implemented. Later robot, target, and obstacle movement were added into the simulation. The GAP had problems in this scenario such as finding in-optimal solution and colliding with obstacles. A collision avoidance system was developed but it had some design flaws. This was the current state of the project when it was handed to Path Planners.

Path Planners goals in the project were to improve upon the collision avoidance system, to find methods of improving the existing GAP's performance, to create and compare a new path planning algorithm, and to implement a radius of vision system to approach a more realistic simulation.

1.2 Objectives

Path Planners based their proceedings on the following set of goals:

1. Investigation of the old collision avoidance system and address its problems and either fix its problems or design and implement a new collision avoidance system that solves the problems of the old one.

2. Investigation of methods of improving the system performance.

   Locate bottlenecks and various problems and shortcomings in the system. Develop solutions to these problems that will improve system performance.

3. Implement a radial vision system to improve and optimize solutions.

2 Background

2.1 Metaheuristic Optimization Methods

Optimization problems (such as finding an optimal path for a robot to travel along) are often extremely complex. Finding the best solution is computationally expensive, time consuming, and usually not feasible. Metaheuristic optimization methods use simple concepts to search through a solution space and find good solutions quickly. They often mimic naturally occuring phenomena such as genetic evolution (GA), ants searching for food (ant colony algorithm), molecules alligning in metal during annealing (simulated annealing algorithm). These optimization methods can be either population-based (they use multiple solutions in parallel to search through the sultion space) or single-solution-based (they use a single solution to search through the solution space).

Metaheuristics do not gaurantee convergance to the global best solution (except simulated annealing given infinate time), but they have been proven to converge to good solutions quickly with efficient use of computing power.

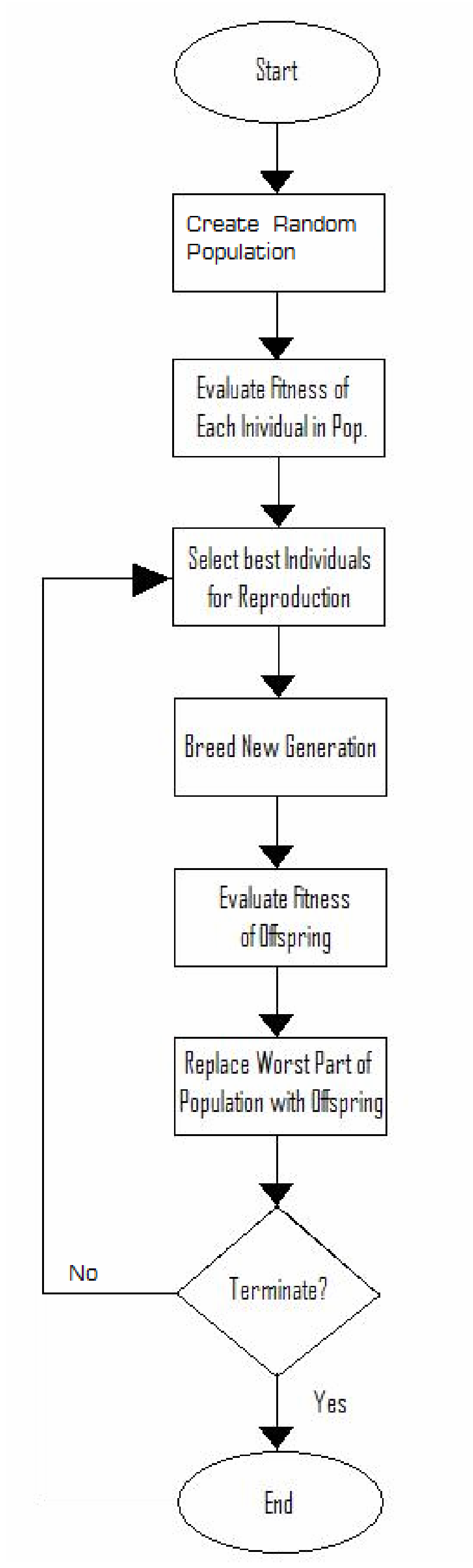
2.2 Genetic Algorithm

A genetic algorithm is one which simulates the genetic competition and evolution of a species in nature. The GA does this by creating a population of solutions, which breed with eachother to generate offspring. Each solution is mapped out as a string of genetic data (For example, in the case of the GAP used in this path planning project each solution was mapped as a sequence of x and y coordinates representing the x and y coordinates of the nodes forming the path). Each solution is evaluated by a fitness function, and solutions with better fitness values are given a better chance of being selected for breeding. Breeding occurs between a pair of parent solutions and results in the generation of a pair of offspring.

Offspring are formed from a recombination of their parent's genes. This is done by taking one part of parent A's genetic data, and combing it with the opposite part in parent B's genetic data (For example, the first three genes of parent A are combined with the last three genes in parent B). This procedure is performed twice (once for each offspring) and uses opposite parts for each offspring. Each offspring is then subjected to mutation, which occurs randomly but does so at a specified probability. This mutation introduces new genetic makeup into the population and allows the algorithm to explore new solutions. The population size is retained by either killing off the old generation (the parent pair) or the worst two solutions from the combination of parent and offspring pairs.

GA's are commonly used in complex optimization problems as they have proved in the past to perform well in these situations. For a detailed explanation of how the GAP implemented in this project functions, the reader should consult the thesis paper listed at [3].

The following flow chart outlines the procedure of a GA.



Start
Create Random Population
Evaluate Fitness of Each Inividual in Pop.
Select best Individuals for Reproduction
Breed New Generation
Evaluate Fitness of Offspring
Replace Worst Part of Population with Offspring
Terminate?
No
Yes
End

Figure 1: Flow of GA

2.3 Collision Avoidance System

2.3.1 Purpose of Collision Avoidance

When robot and obstacle dynamism were added into the Path Planning Problem it became clear that a collision avoidance system was needed. The GAP that had initially been developed had been designed to avoid non-moving obstacles only. Though obstacles could be added into and removed from the environment, the obstacles that were in the environment did not move around. Being so, when dynamism was added into the system the robot was unable to avoid the now moving obstacles. As a result, non-optimal solutions and frequent collisions occurred. A simple PCAS (Pre-emptive Collision Avoidance System) was developed and added into the system to solve these problems.

2.3.2 System Overview

The existing PCAS used a collision detection method and a modified fitness function to avoid collision with moving obstacles. Collision detection was performed at each iteration of the GAP by assuming obstacle and robot velocity will remain constant, and for each path looking ahead into the future to see if a collision occurs. Detected collisions were used to penalize the solution that they are detected on through a modified fitness function. The modified fitness function used the summed total of 4 weighted factors: smoothness cost, clearance cost, length, and collision cost. The collision cost is calculated as 1/timeToCollision where timeToCollision is the number of iterations into the future in which the first collision on the path will occur. Collisions occurring further into the future are penalized less heavily than collisions occurring early on.

2.3.3 Problems with the Old Collision Avoidance System

The old PCAS has a few problems. One is that the system sometimes can lead to non-optimal solutions when avoiding collision with moving obstacles as shown in Figure 2. Also the system can potentially get stuck in an endless collision avoidance manoeuvre as shown in Figure 3, never reaching the target destination. The third problem is that the system sometimes guides the robot into a trap where a collision is inevitable as shown in Figure 4. Finally in some situations collisions could be avoided but the system fails to do so as shown in Figure 5.

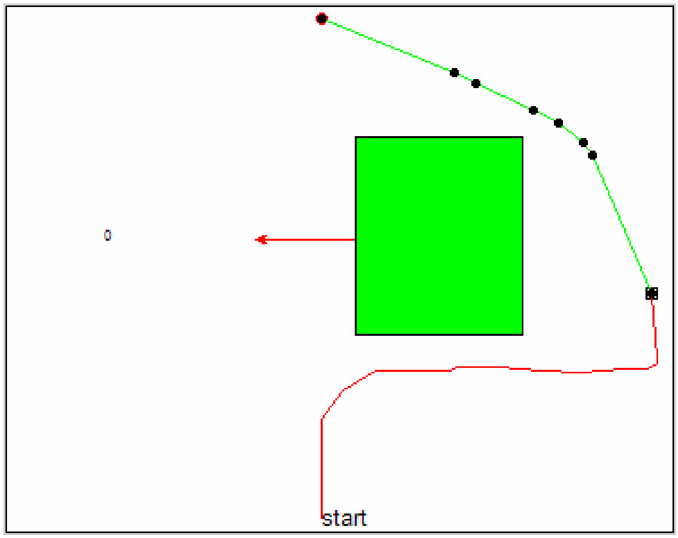


Figure 2: In-optimal Path

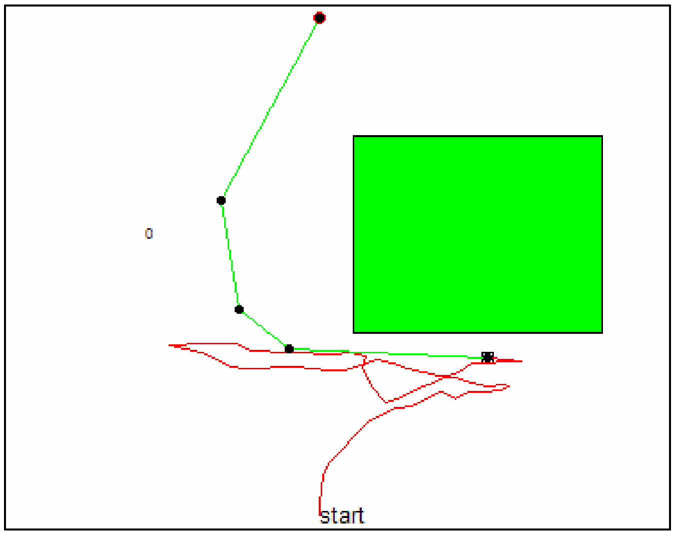


Figure 3: Endless Loop

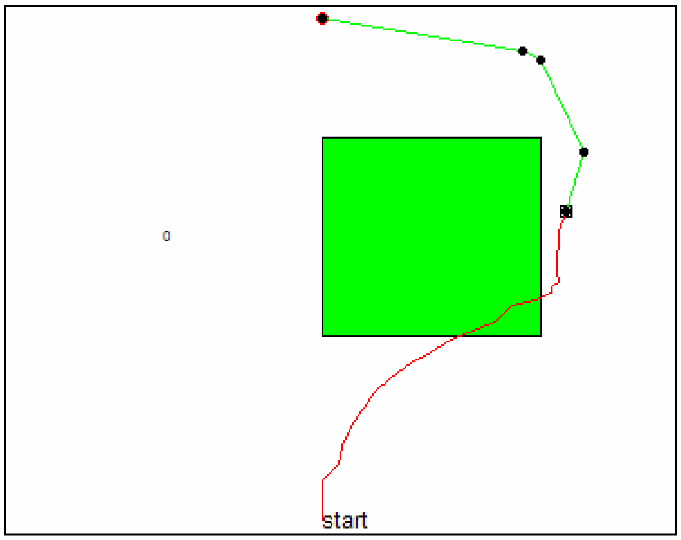


Figure 4: Imminent Collision Due to Trap

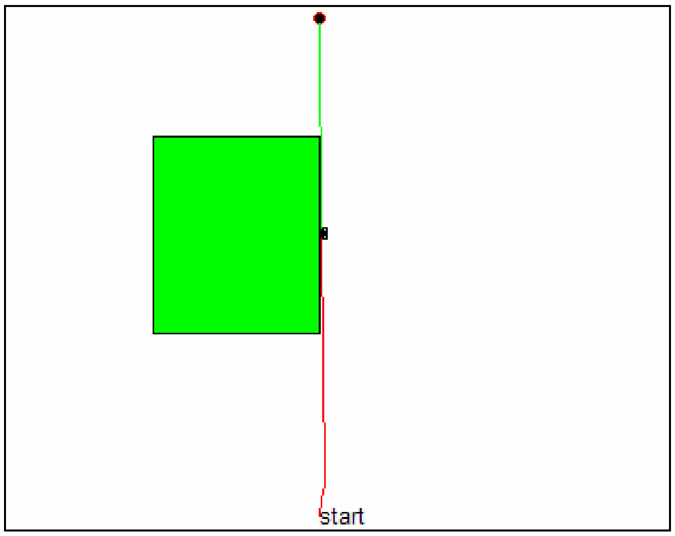
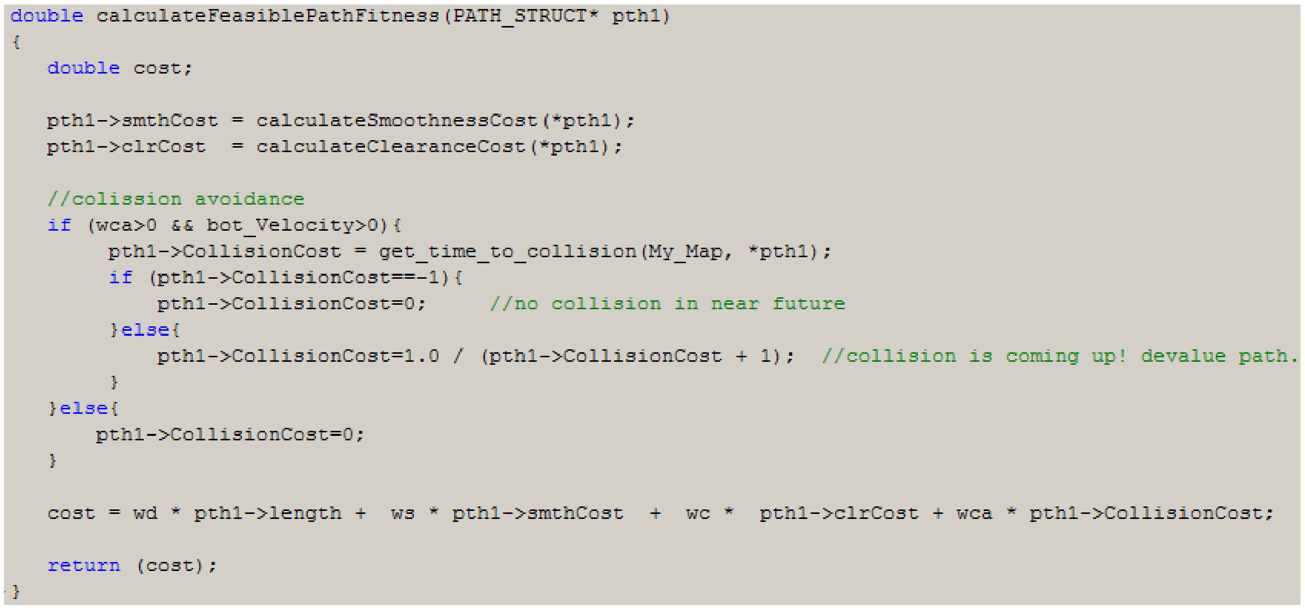


Figure 5: Avoidable Collision

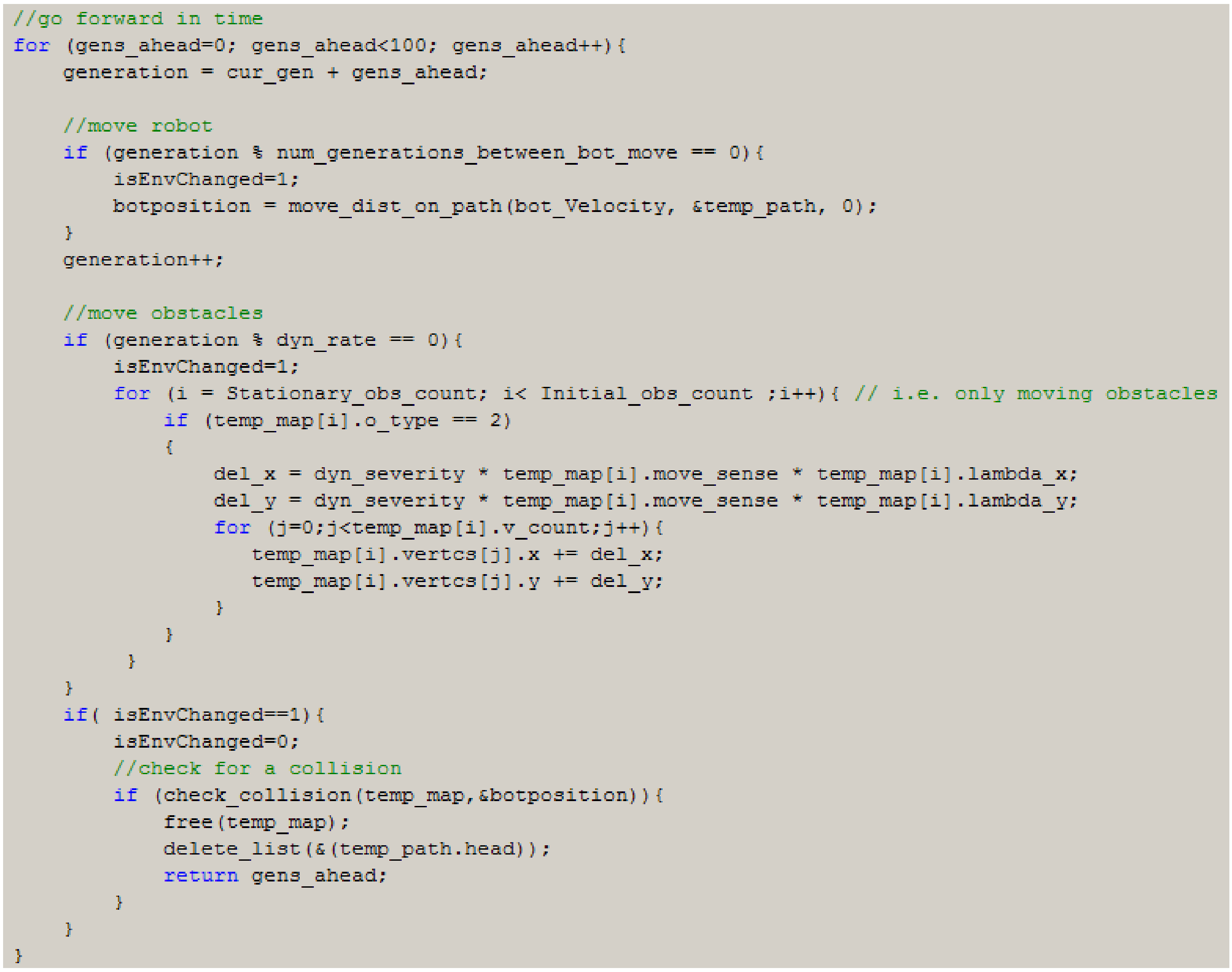
2.3.4 Causes of PCAS Problems

In the early stages of the project timeline much testing was performed and careful observations were made in both test runs and source code in order to determine the root cause of the problems occurring in the old PCAS. Thorough investigation led to several conclusions. The collision detection method and modified fitness function used by PCAS are in fact working correctly and are being employed for each path in the map at each iteration. The collision detection method is called immediately after the calculation of path length, smoothness cost, and clearance cost. The collision detection method performs up to a maximum of 100 iterations and correctly returns the number of iterations that elapsed before the first collision was detected. This number is then used in the modified fitness function to penalize paths that result in collisions, with collisions occurring sooner being punished heavier than collisions occurring later. A code snippet showing where the collision detection method is called and how the value is used in the modified fitness function is shown in Figure 6. This code can be found in path_planning_planner_utils.c in function calculateFeasiblePathFitness(). A code snippet showing how the collision detection method works is shown in Figure 7. This code can be found in path_planning_dynamism.c in function get_time_to_collision().



```
double calculateFeasiblePathFitness(PATH_STRUCT* pth1)
{
   double cost;

   pth1->smthCost = calculateSmoothnessCost(*pth1);
   pth1->clrCost  = calculateClearanceCost(*pth1);

   //colission avoidance
   if (wca>0 && bot_Velocity>0){
        pth1->CollisionCost = get_time_to_collision(My_Map, *pth1);
        if (pth1->CollisionCost==-1){
            pth1->CollisionCost=0;     //no collision in near future
        }else{
            pth1->CollisionCost=1.0 / (pth1->CollisionCost + 1);  //collision is coming up! devalue path.
        }
   }else{
       pth1->CollisionCost=0;
   }

   cost = wd * pth1->length +  ws * pth1->smthCost  +  wc *  pth1->clrCost + wca * pth1->CollisionCost;

   return (cost);
}
```

Figure 6: Fitness Function Calculation



```
//go forward in time
for (gens_ahead=0; gens_ahead<100; gens_ahead++){
    generation = cur_gen + gens_ahead;

    //move robot
    if (generation % num_generations_between_bot_move == 0){
        isEnvChanged=1;
        botposition = move_dist_on_path(bot_Velocity, &temp_path, 0);
    }
    generation++;

    //move obstacles
    if (generation % dyn_rate == 0){
        isEnvChanged=1;
        for (i = Stationary_obs_count; i< Initial_obs_count ;i++){ // i.e. only moving obstacles
            if (temp_map[i].o_type == 2)
            {
                del_x = dyn_severity * temp_map[i].move_sense * temp_map[i].lambda_x;
                del_y = dyn_severity * temp_map[i].move_sense * temp_map[i].lambda_y;
                for (j=0;j<temp_map[i].v_count;j++){
                   temp_map[i].vertcs[j].x += del_x;
                   temp_map[i].vertcs[j].y += del_y;
                }
            }
         }
    }
    if( isEnvChanged==1){
        isEnvChanged=0;
        //check for a collision
        if (check_collision(temp_map,&botposition)){
            free(temp_map);
            delete_list(&(temp_path.head));
            return gens_ahead;
        }
    }
}
```

Figure 7: Collision Detection Method

This implementation only considers the time to a collision and does account for severity of the collision and it is still possible for infeasible solutions to have better fitness than feasible solutions. Another problem with the old PCAS is that the scaled collision cost in the fitness function does not properly promote convergence to good solutions. Solutions with good genes but an early collision are heavily penalized giving the good genes little chance to survive, whereas solutions containing few good collision avoidance genes but result in a late collision are less penalized and their genes have a better chance of survival. The last problem that was discovered with the old PCAS is that it is unable to properly assess clearance cost on dynamic obstacles. The system assesses obstacle clearance based on where the obstacle currently is in relation to the path, and not on where the obstacle is going to be in relation to the path. These design flaws are the cause of the problems previously mentioned.

2.3.5 Detailed Explanation of PCAS Problems

Collision is Avoidable but the System Still Fails

Referring back to figure 5, this situation occurs when all paths in the population will result in collision. This is the result of either:

1. An insufficient number of nodes such that a collision is unavoidable, or
2. The system cannot find a solution to avoid collision quickly enough.

In the first scenario, the path segment containing the collision is bound by two nodes which are each bound in movement either due to environment restrictions (i.e. moving a node will place it in an infeasible location and devalue the path), or the node is a start or end node and cannot be moved. The path cannot be repaired properly by adding new nodes via the smoothness operator or the repair operator because either no turns or corners exist to be smoothed, or adding nodes with the smoothness operator causes the path to become heavily devalued, or because the repair operator will not create new nodes until the obstacle enters the path segment, however at this time a collision has already occurred. This situation is depicted graphically in Figure 5.

In the second scenario, a collision is detected on every path and cannot be avoided with the current number of nodes. The system cannot form feasible paths until new nodes are added. New nodes are eventually added, but it is too late for them to become effective or they are not in the right areas. The repair operator will not add new nodes until an existing node is moved such that the existing path becomes infeasible. Such a move would cause the path to be devalued and the path's genes would have a low chance of surviving. The smooth operator will only add new nodes to smooth out corners. If no sharp corners exist, or if smoothing existing sharp corners causes poor path fitness (due to clearance, length, or collision costs) the solution has a low probability of surviving in the population’s gene pool. With low probability of survival, these genes are often quickly lost, and may not survive long enough to mutate and crossover operators to move new nodes into areas that would avoid collision.

Both scenarios would be solved if new nodes were added to all the paths where a collision will occur so that these genes have a good chance of survival. Also, by adding the new nodes into areas where they are most needed (where the collision will occur) the system would be able to react quickly and form collision avoiding paths sooner.

System Finds Non-Optimal Solutions When Avoiding Collisions

Referring back to Figure 2, this situation occurs in 1 of 4 cases:

1. A lack of population diversity results in convergence to a local minimum as opposed to the optimal solution area,
2. The system is unable to find feasible paths within an appropriate time frame,
3. The collision cost factor in the fitness function does not properly promote convergence to good solutions or,
4. The clearance cost factor of the fitness function is unable to properly assess good solutions.

In the first scenario, limitations on the mutation operator and behaviour of the fitness function can cause the system to become trapped in a local minimum that is a non-optimal solution. This premature convergence may occur because the mutation rate is too low and new genes are dying out faster than they are being introduced, or because the maximum changes allowed to be made by mutation are too small to break free of the local minimum. For example, in the situation depicted in Figure 2, if the smoothness operator were weighted less heavily, then it may have encouraged the creation of solutions which did a 360 degree turn and went on to avoid the obstacle from the left hand side as shown in Figure 8.

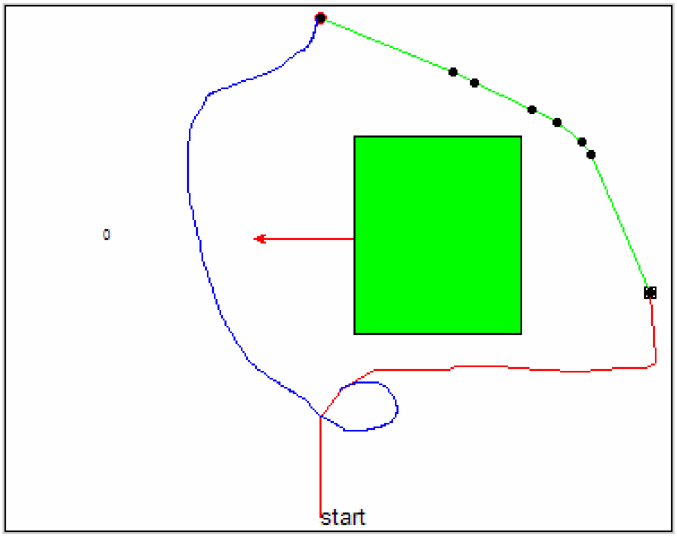


Figure 8: Possible new solution

In the second scenario, the problem is similar the issue in which the system detects collisions on all paths but is unable to form collision avoiding paths soon enough or at all. In Figure 4 we can see that the system was unable to form collision avoiding paths until the obstacle crossed into the path of the robot and made the path infeasible, forcing the repair operator to add new nodes and allowing the system to form collision avoiding paths. Late avoidance of collisions leads to the first scenario in which the system becomes trapped in a local minimum that is a non-optimal solution.

In the third scenario, failure to convergence to good solutions lies in how the collision avoidance system analyzes solutions that result in collisions, and how they are penalized. PCAS is only able to recognize collisions with obstacles and when they will occur. It is unable to recognize which parts of the path are the major causes for collision. Also, there is a good chance that situations occur when two very similar paths are given very different fitness ratings because one path has an early collision whereas the other has a late collision. This unbalanced fitness rating leads to poor convergence to good solutions and can be a major cause of in-optimal solutions.

In the fourth scenario, the problem with the clearance cost is that when dealing with moving obstacles the system is unable to properly use the clearance cost factor in the fitness function. This factor plays an important role in converging to good solutions, and when dealing with dynamic obstacle it works very poorly if at all. The clearance cost is calculated based solely on the current location of the obstacle in relation to the path. This is clearly flawed since in fact the system should be looking at where the obstacle is going to be in relation to the path. A path that has good clearance now may be completely covered by the obstacle in the next few iterations. An opposite example would be a path that has bad clearance now, but may be very far away from the obstacle in several iterations. By ignoring or improperly analyzing these scenarios the convergence of the GAP is greatly reduced.

Possible solutions for the first scenario would be to adaptively increase mutation rate and increase maximum distance of node position changes made by mutation when collisions are detected. Also, having an adaptive fitness function that reduces penalties on smoothness when collisions are detected could help the system escape from local minima. Solutions for the second scenario are the same as those for the case in which the collision is avoidable but the system still fails. For the third scenario, a possible modification may be to calculate for how many iterations the robot is colliding with or is inside the obstacle and devalue the path fitness according to this value. In this way, paths that result in many collisions will be more devalued than paths with few collisions, and solutions will converge to optimal sooner. For the fought scenario, a similar solution can be taken as for the third scenario. For dynamic obstacles, clearance cost can be taken as an average of how close the robot comes to the obstacle at every iteration into the future. This would begin to approximate the proper behaviour of the clearance factor, but not be an exact solution.

System Guides Robot Into A Trap

This situation is an extension of the problem where the system leads the robot into a non-optimal solution, where the non-optimal path found by the system is in fact a solution that will lead to an imminent collision. The causes for this are also extensions of the same problem.

Solutions for these issues are covered by the solutions to the problem described above.

System is Stuck in Endless Avoidance Loop & Never Reaches Target

This situation is a form of the non-optimal solution problem described above in which the non-optimal solution is one that never reaches the target, essentially rendering it infeasible. In this case, the system is able to escape from the local minimum that is a non-optimal solution, but does so too late and falls into another local-minimum. The system switches from local-minimum to local-minimum at the same frequency that the obstacle is switching directions. In short, the system determines that the best solution is to change direction and go around the moving obstacle at the same time that the obstacle itself changes direction.

Possible causes are similar to those for non-optimal solutions that reach the target but with poor population diversity. Limitations in the mutation operator and the fitness function make it difficult for the system to break free of local minima. However, in this case the system is eventually able to do so, but not early enough. Possible causes for lateness in breaking free of local minima may be that the collision cost does not reach a high enough value until the obstacle nears a wall or another obstacle, or that the node mutation direction converges to one direction once the obstacle nears a wall.

As a moving obstacle approaches a wall or another obstacle, the area between the two obstacles or the obstacle and the wall will grow smaller and smaller. As it does so, the possible paths will be forced to come closer and closer to the obstacle they will collide with. As this happens, the timeToCollision will grow smaller and smaller thus increasing the collision cost. Once the cost is high enough the system is able to break free of the local minimum.

When the obstacle nears a wall, the possible paths will get closer to the wall as well. When this happens, repairs and mutations become infeasible as they place nodes outside of the feasible range. This will slowly force the direction of mutation to converge away from the wall, which assists the system in breaking free of the local minimum.

Possible solutions for these problems are covered by the solutions to the problem of finding non optimal solutions as well as to restrict mutation direction on certain nodes in these types of situations.

2.3.6 Summary of Possible Solutions to Problems with PCAS

This section summarizes the possible solutions to problems in PCAS mentioned in the previous section. The major problems with PCAS are its inability to add new nodes to paths except via the repair and smooth operators, its improper use of the clearance cost, and its flawed calculation of collision cost. One possibility for adding new nodes would be to add a new mutation operator. This mutation operator would be responsible for adding new nodes to each path. The rate of mutation could be adaptive and also be unique to each path. Thus, paths with collisions detected could have a huge increase in this mutation rate and have a sudden increase in nodes. The downside in this method is that it would not guarantee that the nodes would be place where they are needed. A more proper calculation of collision cost could be formed by calculating the number of iterations into the future that the robot would be colliding with or be inside the obstacle. This value would then be used as the collision cost. This method would be a more accurate way of determining which solutions contain good collision avoidance genes and would promote convergence to these areas of the solution space. A better method of calculating clearance cost when dealing with moving obstacles could be formed by calculating the distance from the robot to the obstacle at each iteration into the future. These distances could be summed, averaged, and then added to the clearance cost for the rest of the path (the static objects). This would cause the clearance cost to behave more closely to how it was intended, and promote solution convergence to good solutions.

Solutions to less major problems and possible improvements for performance with PCAS are to introduce adaptive mutation rate, adaptive maximum node movement distances due to mutation, and adaptive fitness function weights such as devaluing smoothness, all to help escape local minima and avoid infeasibility.

2.4 PSO

Particle Swarm Optimization is a population-based met heuristic optimization method in which solutions communicate with each other and work together to find optimal solutions. The method simulates the behavior of a swarm of a certain species working together to reach a common goal (For example, a school of fish locating food on the ocean floor). Solutions are given a position and velocity in the solution space, and have an inertia associated with their movement. Solutions are divided into neighborhoods, and each solution is aware of the best position it has visited during the search (its local best), and also the best position that its neighborhood has visited (global best). Neighborhood size can vary and can even be equal to the size of the population (the whole population is one neighborhood).

At each iteration the velocity vector of each solution is updated and is the result of combining three vectors:

1. Its current velocity vector
2. The vector from its current position to the position of its local best
3. The vector from its current position to the position of its global best

These vectors are combined in varying strengths (ie. Each one has a weight). Specifically, the formula by which these vectors are combined is as follows:

$$v_{new} = \alpha \bullet v_C + rand(0,\beta_L)(p_L - p_C) + rand(0,\beta_G)(p_G - p_C)$$

α = weight of inertia

β = weight of Local or Global Best

v = velocity

p = position

Subscripts: L = Local Best, G = Global Best, C = Current

The resulting vector is used to update the position of the solution in the solution space.

$$p_{new} = p_C + v_{new}$$

The new position of each solution is then evaluated and compared to its local and global best. If the new position's fitness is better than the local or global best position, the local or global best is updated accordingly.

PSO algorithms, like GA's, are commonly used in complex optimization problems since they also have been shown to perform well in these situations. A detailed explanation of how PSO was implemented into this project is outlined in the design section of this report.

3 Design Methodology

3.1 Collision Avoidance System

A few solutions to the problems with the collision avoidance system were developed that all approached the situation from the same perspective. The general idea being avoid the problem before it is a problem. The interim report proposed a complicated solution that extrudes the area of an object in the direction of its velocity and analyses paths through this extruded area. (See the appendices for an overview of the Intelligent Collision Avoidance System (ICAS)). This system inspired discussion of a table based system that keeps track of the status of each path for a number of iterations into the future. The purpose of this would be to devalue paths based on the number of collisions they experience as well as how soon collisions occur.

The idea that some paths ending in a collision could potentially have better fitness than feasible paths was also discussed and the collision scaling routine was developed as a method to alter the fitness values assigned to each path based on the fitness values of the other paths in the solution space.

3.1.1 Preliminary Collision Scaling

The collision scaling routine was implemented as a cost evaluation function meaning that it alters the fitness of paths after the existing system calculates and assigns fitness values for each path. By implementing the new collision avoidance system as a post fitness calculation evaluation, the system accommodates integration with both GA & PSO algorithms. The collision scaling routine is designed to alter path fitness of only the paths experiencing collisions with dynamic or static obstacles.

The routine scales path fitness such that all feasible paths have better fitness values than any path resulting in a collision and all paths resulting in collisions with dynamic obstacles have better fitness values than any path resulting in a static collision. The idea being that collisions are worse than inefficient paths and dynamic obstacles might move where as static obstacles will never move out of the way.

The function works by scanning the list of paths to determine the worst case feasible path fitness and best case dynamic path fitness, and then uses the following equation to alter the fitness of all paths resulting in dynamic collisions.

```
Path Fittness = Paths Fittness + (Feasable Path Worst Case Fitness
            - Dynamic Colision Best Case Fitness) + 1;
```

The effect of this equation is that the path fitness of all paths resulting in dynamic collisions are at least 1 fitness point worse than the worst case fitness value of the feasible paths.

The process is repeated for paths with static collisions with respect to the worst case fitness of paths resulting in dynamic collisions.

The type of collision is determined by the get_future_violations() function explained in the next section. The collision scaling function is shown in the appendices of this report.

The cost_evaluation() function is called at the end of the computePopFitness() function.

3.1.2 Obtaining Time to Collision & Type of Collision

Collision scaling, as well as other functions in the cost evaluation routine make use of a function designed to store the number of collisions a path incurs as well as the type of obstacle the path collided with within the path structure of each path. The get_future_violations() function is shown in the appendices of this report and the following paragraphs outline its methodology.

The get_future_violations() function works on this premise: create a copy of the map and try each path on it for a number of iterations and report back how each path performed before proceeding on the actual map. The copied map performs object moves just like the real map and runs for 500 iterations. For each iteration the function examines if the robot is in a collision and what type of object the robot is colliding with. If the robot does experience a collision a point is added to a FutureViolations variable and the obstacles type is stored. If the robot experiences no collisions a 0 is stored, if the robot experiences a collision with a dynamic obstacle a 1 is stored, and if the robot experiences a collision with a static obstacle a 2 is stored which cannot be reverted to a 1 (i.e. once a path is deemed to have collided with a static obstacle this will not be forgotten).

The actual collision type is determined in the check_collision_and_type() function which works by checking a given point on a given map with the is_pointInPolygon() function previously implemented. The check_collision_and_type() function is shown in the appendices.

3.1.3 First Violation and Future Violations Evaluation

The next two functions work in a similar manner to determine within the paths resulting in collisions, which paths have later collisions and which paths have the least number of collisions; these conditions being favourable over paths with collisions sooner in time and with more collisions. These functions maintain separation by type of collision and require the path list to be sorted into the types of collisions (0 = no collision, 1 = collision with dynamic obstacle, & 2 = collision with static obstacle). A bubble sort function was implemented to handle this procedure. Within this bubble sort the capability to swap fitness values between collision types was added such that when a path with a higher valued collision type swapped places with a path with a lower valued collision type the fitness could be swapped as well if it was found that the path with the higher valued collision type had a better fitness than a path with a lower valued collision type.

The FutureViolations function similarly employs a bubble sort within each violation type swapping fitness values when a path with more collisions has a better fitness than a path with fewer. Similarly the FirstViolation function swaps fitness values when a path with a sooner first collision has a better fitness value than a path with a later first collision.

3.2 Radius of Vision

Below is a typical dynamic environment where there are five static obstacles (yellow), and two dynamic obstacles (green). The dynamic obstacles are moving in the direction shown to the threshold of the map and then reversing in direction.

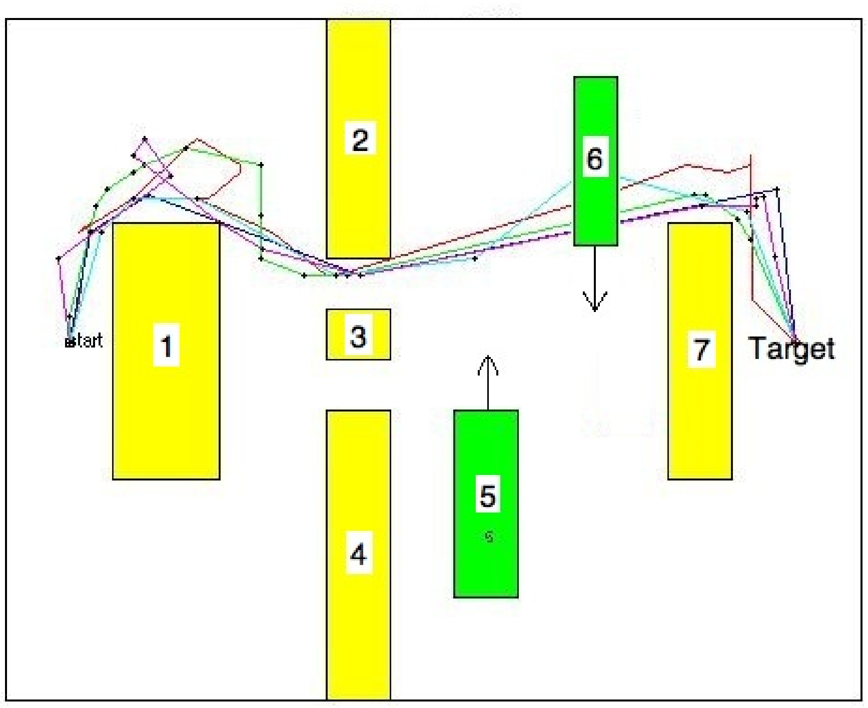


Figure 9: Initial Map Layout

Under the current implementation of the program, all paths would be assigned a fitness value. Each path is evaluated based on its length, smoothness, and possible collisions with obstacles. In the case of the figure above, the paths in the portion of the population shown are initially intersecting with obstacle six. This would result in a poor fitness value and the probability of these paths surviving and evolving is very low. However, some time t later, the map could look as it does below.

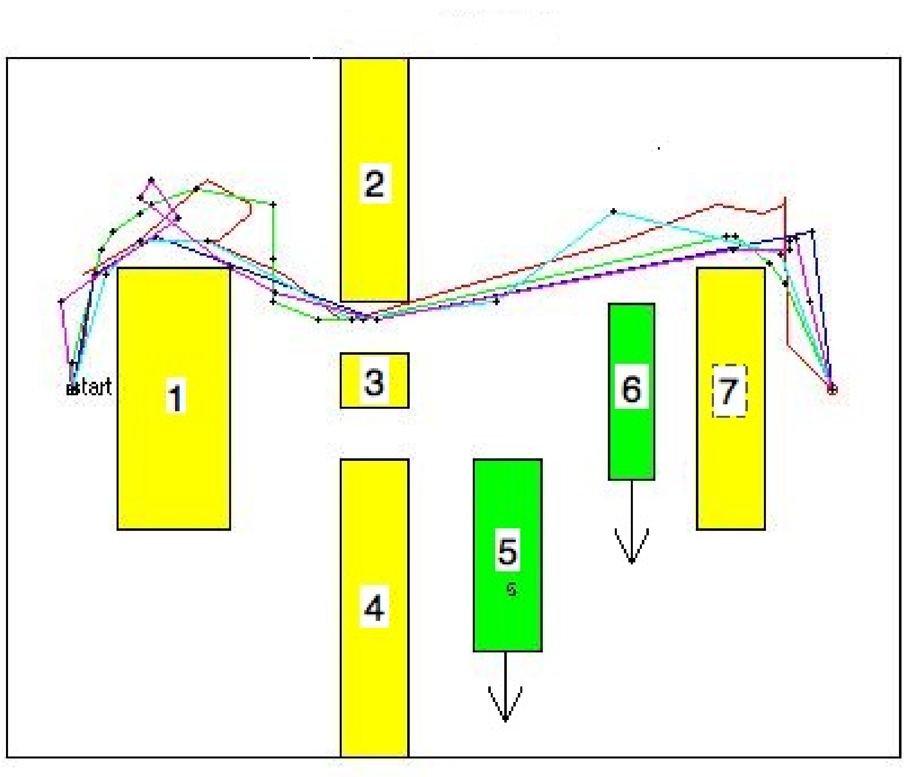


Figure 10: Map Layout Time t in the Future

Obstacle six has now moved and the path population shown has become feasible, as there are no longer any intersections. To avoid situations like this one, a radial vision system has been introduced. Now all paths in the population are evaluated based solely on a portion of the overall path. Essentially, this method limits the robots line of sight evaluating the paths in close proximity or radius of the robot. In complex dynamic environments, evaluating the entire path at each iteration is a waste of resources since the map is changing continuously.

3.3 Implementation

Initially, without radial vision the program populates a linked list called pathsPop() containing all of the paths under evaluation at each iteration. Radial vision was implemented by creating a secondary list that contained only a given number of nodes for each path in the population. In the example below the robot is on route to the destination and there are five paths under consideration.

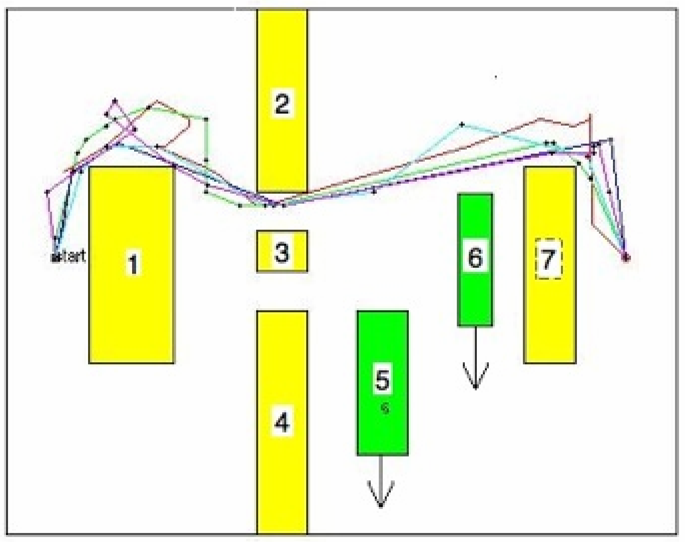


Figure 11: Radial Vision Implementation

These paths are evaluated based on a given number of nodes that is passed as a command line argument. The user therefore has the option of how many nodes are used for path evaluation. Since the number of nodes in each path varies, a second method of radial vision was implemented. Instead of passing a hard coded number of nodes to be evaluated, a percentage of all nodes in the path were evaluated. For example, if the user wanted to use ten percent of the nodes for evaluation, then that percentage is used in place of a hard coded value. This method was thought to be advantageous since the paths not only contain different numbers of nodes, but those numbers are potentially changing with each iteration. The results of testing these two implementations will be discussed later in this report.

In programming radial vision, there was one method added to path_planning_planner_utils.c; createRadialPath(). This method receives a pointer to a single path in pathsPop(), and also the number of nodes to be evaluated. It then creates a new path based on the supplied number. The following is the snippet of code that is responsible for this.

```
while (current != NULL && numNodes != 0){
        if (radialList == NULL){

            Push(&radialList, current->x,current->y);
            radialList->segClr = current->segClr;
            radialList->nodeAngle = current->nodeAngle;
            radialList->segIntersections = current->segIntersections;
            radialList->next = NULL;
            tail = radialList;
        }
        else {
             //tail->next = malloc(sizeof(NODE));
             Push(&tail->next, current->x,current->y);
             tail = tail->next;
             //tail->x = current->x;
             //tail->y = current->y;
             tail->segClr = current->segClr;
                             tail->nodeAngle = current->nodeAngle;
             tail->segIntersections = current->segIntersections;
             tail->next = NULL;
         }
         current = current->next;
               numNodes--;
    }
```

The code above traverses the path copying each node to the new list until the number supplied has been copied. The function then goes on to evaluate the new path before returning its fitness. The program can be run both with and without radial vision depending on the command line arguments that are supplied. The +Rad flag followed by a number will cause the program to execute using radial vision with the specified number of nodes. Providing no flag will default to normal program execution.

When the robot reaches the last 25% of the path, the program reverts to normal Path Vision. The reason for this is that there is no need for a fixed radius of vision when there are only a limited number of nodes remaining in the path. It would be a waste of resources to continue creating a new radial path for evaluation.

3.4 PSO

3.4.1 Overview

Due to the limited time and large complexity of the project, steps were taken to simplify the implementation of a PSAP. It was decided to keep the number of nodes in each path constant throughout a run, and equal amongst the population. This simplified the process of comparing particle positions and velocities, as comparing a path with three nodes to a path with six nodes was found to be quite a design challenge and no seemingly good solution was invented by the team. This simplification however, created challenged in designing shortcut, repair, and smoothen operators (operators that can more easily be implemented with a dynamic number of nodes). Another simplification taken was treating the entire population, as a single neighborhood in which each particle's global best was simply the population global best. This simplification was taken since it reduced the complexity of the code, and PSO algorithms are known to have low sensitivity to variations in neighborhood size. Also, the PSAP uses the same fitness evaluation function as used by the GAP, which does not always suite it well. The final simplification taken was to only ensure and test functionality of the PSAP in a static environment. The PSAP is still able to function in an environment in which the robot is given mobility, but only until the robot crosses over a node in its chosen path.

The steps taken by the PSAP during a run are as follows:

1) PSO paramaters are set equal to defaults and updated to command line argument values (if command line arguments are given for the paramaters)

2) Population is initialized with new paths and their local bests are set along with the population global best

3) Population is repaired and shortcutted until all paths are feasible

4) Paths are evaluated and the global best and local bests are updated

5) Each path's (except path 1) velocity and position are updated

6) Shortcut and Disperse Nodes operators are applied at specified rates

7) Steps 4 to 6 are repeated until a fixed number of iterations is reached

Each operator and process used in the steps listed above is explained in detail in the sections following.

3.4.2 Mapping Solutions as Particles

Each path from the robot to the target is treated as a solution. Each path contains a linked list of nodes and a pointer to the start and end nodes. Each path also stores information about its length, smoothness cost, clearance cost, fitness value, number of violations, nuber of nodes, a velocity vector, and a local and global best solution. Rather than storing the local and global best solutions as paths they are stored as pseudo paths, which contain only information about node coordinates and path fitness value. This is the only information that the PSO algorithm must have access to and manipulate when referencing the local or global best solutions. Local and global best solutions are stored as pseudo paths to conserve memory usage.

Each path is considered a particle and is treated as a collection of sub-particles, where each sub-particle represents a node in the path. Each sub-particle's position is the x and y coordinates of its corresponding node. Velocity is the x and y displacement occuring at each iteration. Each sub-particle does not have a local or global best fitness of its own, but instead references the local or global best fitness of the path.

Each sub-particle must have a local and global best position, and a velocity. To make mapping solutions as particles simpler, it was decided to keep the number of nodes in each path constant and equal throughout the population. This allowed each sub-particle's local and global best position to simply be the position of the corresponding sub-particle (node) in the particle's (path's) local or global best particle (psuedo path). This is depicted graphically in Figure 12 below. Sub-particle velocities are the sub-particle's x and y displacement at each iteration.

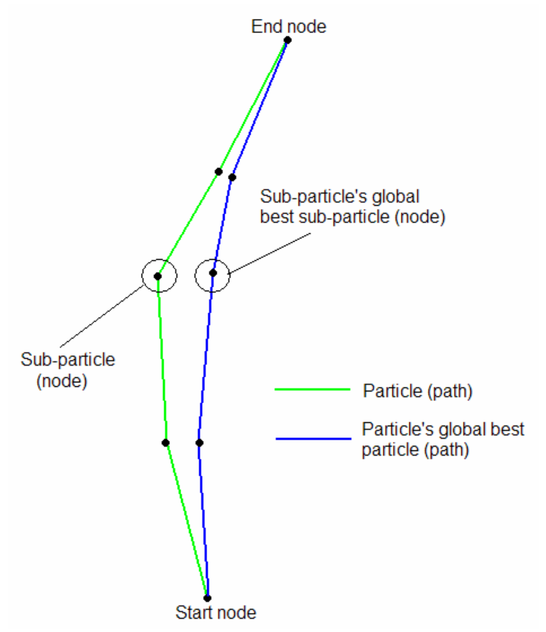

End node
Sub-particle's global best sub-particle (node)
Sub-particle (node)
Particle (path)
Particle's global best particle (path)
Start node

Figure 12: How Paths Are Mapped As Particles

Though keeping the number of nodes constant and equal throughout the population simplified mapping the solutions as particles, it made other aspects of implementing a PSO path planning algorithm more difficult. Where the GAP could perform repair, shortcut, and smooth operators by removing or adding nodes on the fly, the PSAP had to perform the same operations by reconfiguring node positions. This proved to be difficult in the repair operator and smooth operator (disperse nodes operator) of the PSAP which are explained in detail in the following sections.

3.4.3 PSO Parameters

PSO uses several parameters which can be configured in the command line arguments of the program. Each parameter is listed below with a brief description of its function beside it.

Number of Nodes – The number of nodes appearing in each path.

Maximum X Velocity – The absolute value of the maximum X displacement that a sub-particle may undergo at each iteration.

Maximum Y Velocity – The absolute value of the maximum Y displacement that a sub-particle may undergo at each iteration.

Inertia – The effect of inertia on each sub-particle's resulting velocity vector.

Local Best Attraction – The sub-particle's strength of attraction to its local best sub-particle.

Global Best Attraction – The sub-particle's strength of attraction to its global best sub-particle.

Shortcut Rate – The average rate at which the shortcut operator will be applied to each path.

Disperse Nodes Rate - The average rate at which the disperse nodes operator will be applied to each path.

3.4.4 Initialization

For initialization *n* Paths are generated with *m* nodes, where *n* is the number of paths in the population (specified in command line arguments) and *m* is the number of nodes in each path (specified by PSO parameter "Number of Nodes"). The coordinates of the start node and end node in each path are the same for each path and are set as specified by the map file. All other nodes are given random coordinates (that lay within the map space).

Each path's local best and the population's global best node positions are stored in 2-D dynamically allocated arrays that are Number of Nodes long and have a width of two. Each of the columns in the arrays corresponds to a node, with column 1 corresponding to the start node, and column *n* corresponding to the end node. Row 1 represents the nodes' x coordinate values and row 2 represents the nodes' y coordinate values. For each path's local best these values are initialized to the x and y coordinate values of the path's nodes. For the population's global best these values are initialized to the x and y coordinate values of first path's nodes. Each path's velocity vector is stored in a similar manner with a 2-D dynamically allocated array that is Number of Nodes long and has a width of two. Each position in the array represents a node's x and y velocity. These values are initialized to random numbers between the range of (–Maximum X Velocity, Maximum X Velocity) and (-Maximum Y Velocity, Maximum Y Velocity).

After generating the paths and initializing each path's local best and velocity vector, check_population_feasibility_twoObstacles() is called to determine which paths contain infeasible segments. An infeasible segment is one, which crosses through an obstacle. Paths with infeasible segments are then repaired by the repair operator. Each call to the repair operator is followed by a call to the shortcut operator, and then a call to check_feasibility() which calculates the exact number of violations in the path. Paths are repaired until zero violations occur in the path. If the path contains violations after repairing has been attempted 50 times the path is regenerated with new randomly placed nodes, and this new path is then repaired. This is done to resolve a live-lock situation in which the path contains node positions that cannot be repaired by the repair operator (exact cause of this situation is unknown and further investigation must be done to resolve correctly).

After repairing is completed path fitness values are evaluated by calling evaluatePopulation(). updateLocalGlobalBests() is then called to set the local and global best fitness and position values for each path's local and global best.

3.4.5 Repair Operator

Repairing paths without the ability to add or remove nodes proved to be a difficult task. Unlike the GAP, which is able to repair infeasible segments by adding new nodes to the segment, the PSAP had to make repairs by only rearranging the existing nodes. Designing a quick and intelligent method of doing this proved to be one of the most difficult parts in implementing the PSO algorithm into the system.

Each time the repair operator is called it locates the first infeasible line segment formed by two feasible nodes. If one of the nodes in this segment is the path start or end node, nodes are reconfigured so that this segment becomes feasible, and the next or previous segment becomes the infeasible segment. This ensures that the infeasible segment is formed by two nodes, which can be moved. Since the start and end node cannot be moved (unless the robot or target move) this condition must be checked for and repaired before any other repair is performed. The exact method by which this is performed is best summarized by the pseudo code, which follows and also

Figure 13:

```
IF (Node1 is path start)
    Node after Node2 coords = Node2 coords
    WHILE (violations exist between Node1 and Node2)
        Node2 coords = coords of point halfway between Node1 and Node2
    END WHILE
ELSE IF (Node2 is path end)
    Node previous to Node1 coords = Node1 coords
    WHILE (violations exist between Node1 and Node2)
        Node1 coords = coords of point halfway between Node1 and Node2
    END WHILE
END IF
```

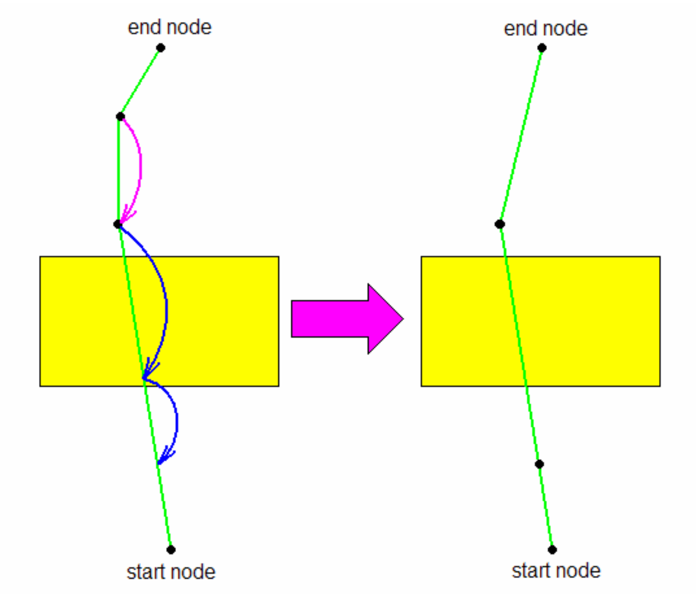


Figure 13: Moving Node Towards Start Node

Next, the repair operator detects if the infeasible segment violates just one or multiple obstacles. If multiple obstacles are violated repairMultiObstacleViolation() is called to reduce the number of violated obstacles to just one. The goal is to have each infeasible line segment violate only one obstacle before the segment is made feasible. This helps to place nodes where they will be needed most, creates more diversity in the solutions after they have been repaired, and avoids attempting to repair certain segment configurations that would be impossible to repair otherwise. Situations in which repair would be impossible with multiple obstacle violations will become apparent after understanding completely how the repair operator works.

Multi-obstacle violations made by segments are repaired by pulling node 2 of the segment towards node 1 until only one obstacle is violated. This is done by halving the distance between node 1 and node 2. If node 2 is pulled in too far and either lays inside an obstacle, or the segment between node 1 and node 2 contains zero obstacles then node 2 is moved back half the distance it was pulled in. This process is illustrated in the following pseudo code and also in Figure 14 below. If the node after node 2 is the path end node then the process is reversed, pulling node 1 towards node 2 instead (this is only a partial solution and may cause live-lock. If after pulling node 1 towards node 2 and attempting repair, there exists a multiple obstacle violation between node 1 and the node previous to node 1, node 1 will be pulled towards the node previous node 1 and a multi-obstacle violation may again be created between node 1 and node 2. This process could potentially keep repeating causing live-lock).

```
WHILE (number obstacles violated by segment > 1)
    Node2 coords = coords halfway between Node1 and Node2
    WHILE (number obstacles violated < 1 OR
              (number obstacles violated = 1 AND node 2 is infeasible))
        Node2 coords = coords halfway between Node2 and Node2 previous position
    END WHILE
END WHILE
```

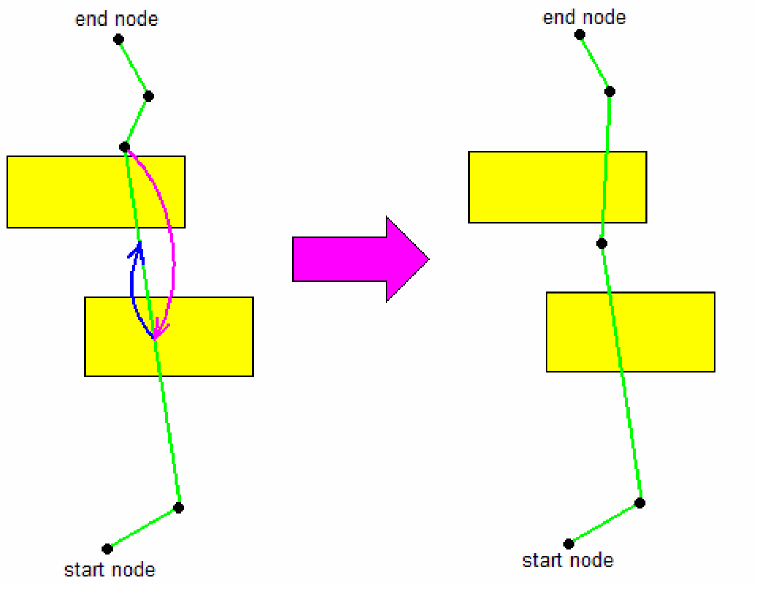


Figure 14: Repairing Multi-Obstacle Violations

Once the first infeasible segment has been found and initialized for repairing, actual repair is done by making random moves in node 1's and node 2's coordinates. The node to move is selected at random, and the x and y displacements to move it by are randomly generated in a range of (-6, 6). Moves that cause the node to be placed in an infeasible location are automatically rejected and new random displacements are selected. Once a feasible move has been generated it is tested to see if it increases the straight line distance from the violated obstacle center to the segment violating it. Moves that increase this distance are accepted while others are rejected. This process is repeated until either the segment becomes feasible or a maximum number of 50 attempts have been made. The concept is to keep moving the segment away from the centre of the obstacle until eventually the segment clears the obstacle. The pseudo code for this process is given below along with Figure 15 depicting the process. If an obstacle lays on a map border its centre point is taken as the midpoint of the edge laying on the map border. Normally the obstacle centre point is taken as the centre point of the smallest rectangle encompassing the obstacle. This repair process will fail on obstacles, which surround the path and require the path to navigate through the obstacle.

```
WHILE (number attempts < 50 OR segment is infeasible)
    randomly select node to move
    move selected node by random displacement between (-6, 6)
    IF (move is feasible)
        IF (distance to obstacle centre did not increase)
            restore node position
        END IF
    ELSE
        restore node position
    END IF
END WHILE
```

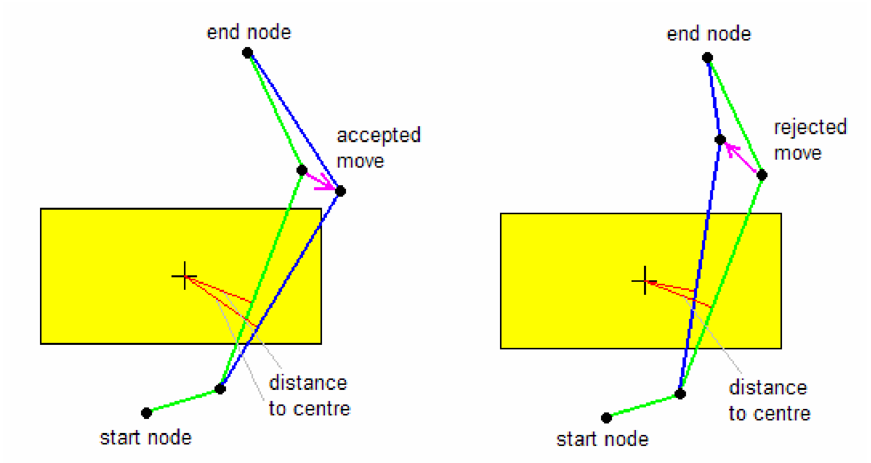


Figure 15: Moving Node To Repair

3.4.6 Shortcut Operator

The PSAP uses a shortcut operator similar to that of the GAP, but rather than removing intermediate nodes, intermediates nodes are repositioned to lay halfway between their encompassing nodes. The shortcut operator is used during repair and then later is used at each iteration at a rate specified by Shortcut Rate. Each call to psoShortcut() makes as many shortcutting node placements as it is able to for the specified path. The operator selects sets of if three consecutive nodes and starts with the set beginning at the start node in the path. Shortcutting is only attempted on the set of nodes if the start and end node of the set are feasible. An attempt to shortcut the set of nodes involves placing the middle node of the set directly halfway between the set's start and end nodes. The move is only accepted if the resulting two line segments formed by the set contain zero obstacle violations, otherwise the move is rejected. After each shortcut attempt the operator selects a new set of three consecutive nodes. The new set starts with the old set's middle node, and ends with the node appearing after the old set's ending node. The process of creating sets and attempting to shortcut them stops when the set ending with the path end node has had shortcutting applied to it. The pseudo code illustrating this process is shown below along with Figure 16 which depicts the process visually.

```
generate set starting with path start node
WHILE (last node in set is not path end node)
    IF (node1 is feasible AND node3 is feasible)
        node2 coords = coords of point halfway between node1 and node3
        IF (segment from node1 to node2 contains violations OR
            segment from node2 to node3 contains violations)
            restore node2 coords
        END IF
    END IF
    generate set starting at node2
END WHILE
```

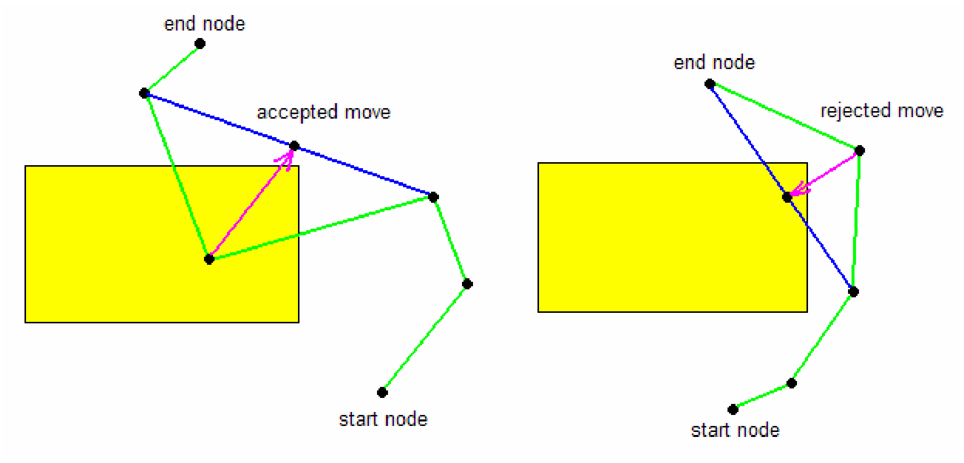


Figure 16: Shortcut Operator

3.4.7 Disperse Nodes Operator

Unlike the GAP, the PSAP is unable to use a smooth operator, which inserts intermediate nodes to add smoothness to a sharp corner. Instead the PSAP must rely on the algorithm's movement of nodes to generate smooth curves. To assist the algorithm in successfully generating smooth corners the PSAP uses a disperse nodes operator which attempts to move existing nodes to where they are needed most. This is done by first selecting the longest line segment in the path and then counting the number of nodes, which exist before and after the segment (not counting the nodes which form the segment). If both the number of nodes before and after the segment are less than two the operator returns. Otherwise the node dispersion begins from the side of the line segment containing the most nodes, or from the front side (side containing the path start node) of the segment if the number of nodes on either side is equal. Node dispersion is performed three different ways depending on the number of nodes before/after the segment. This process will be explained from the scenario in which nodes are being dispersed from the front side of the segment.

Case 1: Exactly two nodes exist before the segment

Let node 2 be the node, which begins the segment and let node 1 be the node before node 2. Node 3 will be the node ending the segment. Node 1 is moved halfway between itself and node 2 and node 2 is then moved halfway between itself and node 3. This is depicted in Figure 17 below.

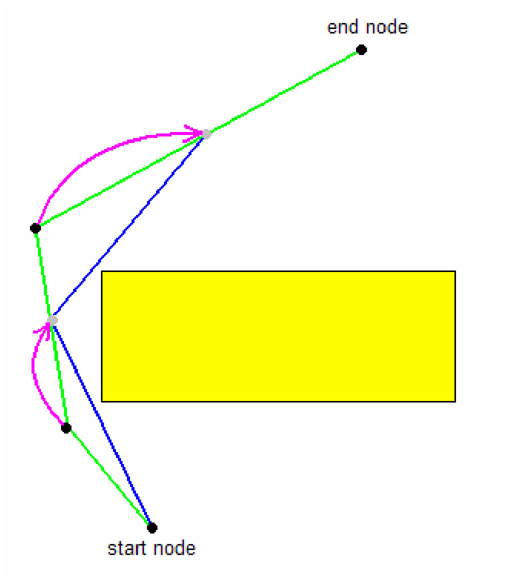


Figure 17: Disperse Two Nodes

Case 2: Exactly three nodes exist before the segment

Let node 3 be the node beginning the segment, node 2 will be the node before node 3, and node 1 will be the node before node 2. Node 4 will be the node ending the segment. Node 1 is moved halfway between itself and node 2, then node 2 is moved to where node 3 is, and then node 3 is moved halfway between itself and node 4. This is depicted in Figure 18 below.

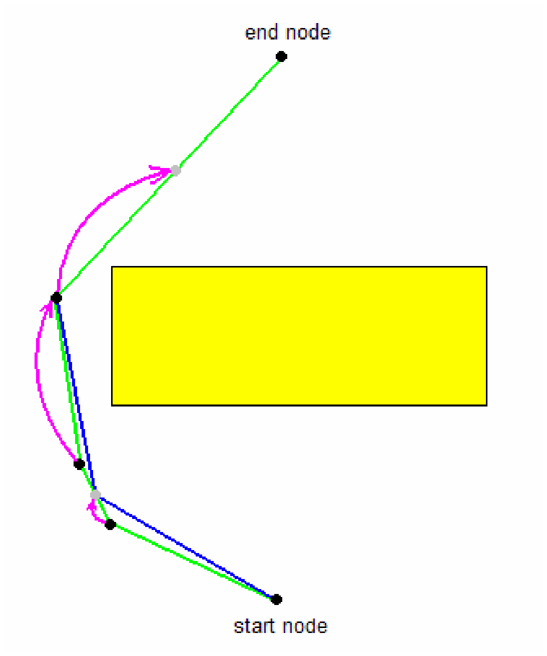


Figure 18: Disperse 3 Nodes

Case 3: Four or more nodes exist before the segment

Let node 4 be the node beginning the segment. Node 3 will then be the node before node 4, node 2 will be the node before node 3, and node 1 will be the node before node 2. Let node 5 be the node ending the segment. Node 1 will be moved halfway between itself and node 2, then node 2 will be moved to where node 3 is located, and node 3 will be moved to where node 4 is located. Node 4 will be moved halfway between itself and node 5. This is depicted in Figure 19 below.

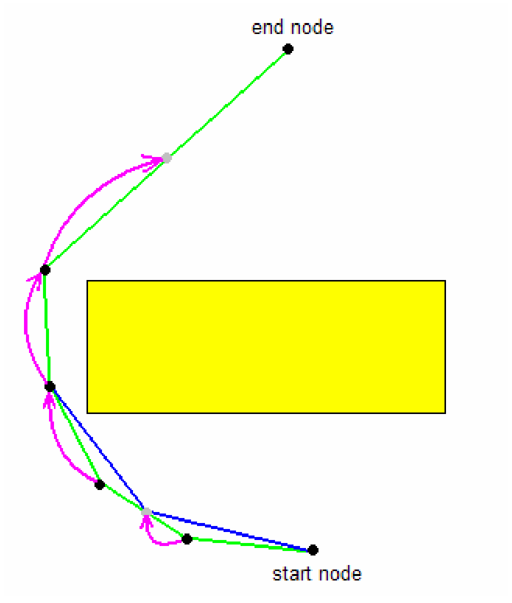


Figure 19: Disperse 4+ Nodes

The disperse nodes operator does not check if the moves it has made have caused any segments to become infeasible. The operator occurs at a rate specified by Disperse Nodes Rate.

3.4.8 Updating Local and Global Bests

For the sake of simplicity the PSAP was implemented with a fixed neighborhood size equal to that of the population. This means that each path's global best is the population best. Also, to ensure that the robot travels down the best path once dynamism is implemented into the PSAP the first path in the population is not updated by the PSO algorithm. Instead this path is maintained equal to the population best solution and is therefore always the best path that the algorithm has discovered, and is the path that the robot will follow.

Each path's local best is maintained as the best (the solution with the lowest fitness value) node configuration that path has seen. The population's global best solution is maintained as the best node configuration any path has seen. Each time updateLocalGlobalBests() is called each path's current fitness value is compared against both its local best and against the population global best. If the path's current fitness value is lower than its local best fitness value, then its local best fitness becomes its current fitness. Also, the x and y coordinate values of the local best solution are set equal to the x and y coordinate values of the current solution. If the path's current fitness value is lower than the population global fitness value, then the global best fitness is set equal to the current path fitness. Also, the x and y coordinate values of the population global best solution are set equal to the x and y coordinate values of the current solution. Also, path 1 in the population is updated to hold the same fitness value and x and y coordinate values as the population global best. This process is depicted in the following pseudo code.

```
FOR (each path in population except path 1)
    IF (path fitness < path local best fitness)
        path local best fitness = path fitness
        path local best node coords = path node coords
    END IF
    IF (path fitness < population global best fitness)
        population global best fitness = path fitness
        population global best node coords = path node coords
        path 1 fitness = path fitness
        path 1 node coords = path node coords
    END IF
END FOR
```

3.4.9 Updating Positions

Each path's velocity vector and node positions (except path 1) are updated at each iteration of the PSAP. This is done inside updatePSOpaths(). The update paths function cycles through each path in the population (except path 1) and starts by updating the velocity vector for each path. The velocity vectors are updated by the equation:

$$v_{new} = \alpha \bullet v_C + rand(0, \beta_L)(p_L - p_C) + rand(0, \beta_G)(p_G - p_C)$$

α = weight of inertia

β = weight of Local or Global Best

v = velocity

p = position

Subscripts: L = Local Best, G = Global Best, C = Current

This equation is applied to each path's nodes (or sub-particles) except the start and end nodes whose positions are dictated by the robot and target positions. The weight of inertia is specified by paramater Inertia, and the weight of local and global bests are specified by parameters Local Best Attraction and Global Best Attraction. After calculation each sub-particles x and y velocity is checked to see if it exceeds Maximum X Velocity and Maximum Y Velocity. Velocities that exceed their maximum allowable limits are trimmed down to the maximum limit. That is, an x velocity great than Maximum X Velocity is set equal to Maximum X Velocity, and an x velocity less than –Maximum X Velocity is set equal to –Maximum X Velocity.

After trimming each sub-particle's x and y velocities are used to update the sub-particle's x and y coordinates. The new coordinates are equal to the old coordinates plus the velocity. That is, sub-particle 1's new x coordinate value is equal to its old x coordinate value plus its x velocity. Coordinates that lay outside the bounds of the map are trimmed accordingly and the corresponding velocity is reversed. That is, if sub-particle 1's new x coordinate value is greater than the map's maximum x dimension then it is set equal to the map's maximum x dimension, and the sub-particle's x velocity value switches sign (negative turns positive and vice-versa).

Sub-particle positions that make the node infeasible are not checked for. The PSAP's performance may be improved by checking for this occurrence and either forcing the sub-particle to skip to the other side of the obstacle, or to bounce off the obstacle with a velocity of equal magnitude and opposite sign. Other methods of handling this occurrence may be necessary to improve the PSAP's performance.

3.4.10 Design Summary and Comments

The PSAP implemented in this project still has much more refinement to undergo and various changes could be made which may improve the PSAP's performance. It may be helpful to change the way in which particle positions are updated when dealing with infeasible particles. Also, the repair operator could be implemented at a specified rate such as the repair operator used by the GAP. It would be very interesting to have the PSAP working in a dynamic environment and to test its performance in this situation. Redesigning a PSAP, which allows for dynamic number of nodes in each path could also lead to improved performance. Finally, many parameters in PSO could be made dynamic and this could have the largest impact on the algorithms performance.

The benchmarks in which the Pap's performance was tested are introduced in the Results section of the report.

4 Results and Discussion

4.1 Collision Avoidance System

Testing of the new collision avoidance system focused on the robots success of reaching the target as well as the CPU time required to reach the target. Testing was essentially performed for fitness altering function by running simulations with and without the function being called.

In all situations the collision scaling function was deemed ineffective despite proof that it was in fact working as planned. This is due to the operation of the elite paths function which chooses the first half of the list of paths to work with. The violation type bubble sort with fitness swapping enabled (which essentially performs the same operation as collision scaling while sorting the feasible paths to the front of the list) was much more effective as will be shown.

Finally in cases where both the FutureViolations & FirstViolation functions were enabled, testing was performed with FutureViolations being called after FirstViolation placing emphasis on the severity of collision, as well as with FirstViolation being called after FutureViolations placing emphasis on how soon a collision will occur.

4.1.1 Benchmark Tests

A series of bench marks were provided with the simulation software to Path Planners. The collision avoidance system was tested on a number of these and the following paragraphs describe the findings.

Benchmark Z1

Three notable cases came from testing on the z1 bench mark. The following images show the base case, a case in which FirstViolation is emphasized by being called after FutureViolations, and a case in which FutureViolations is emphasized by being called after FirstViolation.

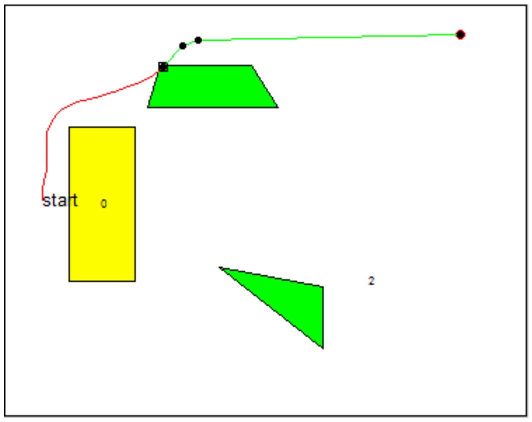


Figure 20: Base Case

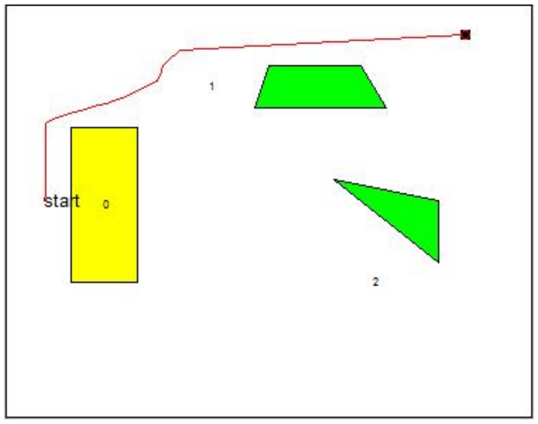


Figure 21: FirstViolation Emphasised

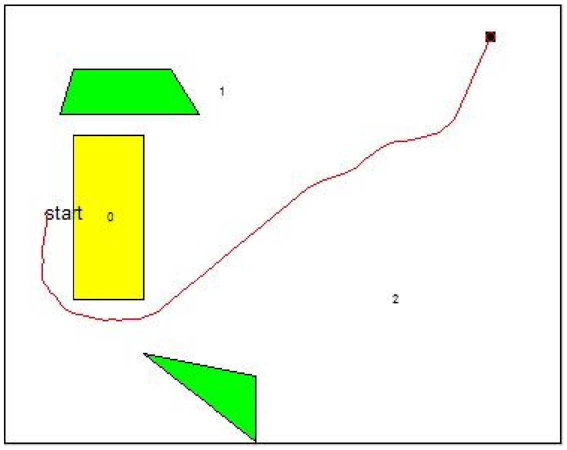


Figure 22: FutureViolations Emphasised

In Figure 20 we observe a collision with the corner of the upper moving obstacle. In the subsequent figures the collision avoidance system prevents this collision in 2 different ways. The FirstViolation emphasised method takes a more vertical approach because this is the more optimal path to follow while avoiding collision. On the other hand the FutureViolations emphasised approach determines that the paths going below the static obstacle are ultimately safer despite the added length of the path.

The following table outlines the success or failure, as well as the CPU execution time of the trials that succeeded.

Parameters	Status	CPU
Collision Scaling	Fail	
Bubble Sort Violation Type (BSVT)	Fail	
BSVT & Swap Fitness (BSVTSF)	Pass	4.196
BSVT & FirstViolation	Pass	4.696
BSVTSF & FirstViolation	Fail	
BSVT & FutureViolations	Pass	4.008
BSVTSF & FutureViolations	Pass	5.538
BSVTSF, FirstViolation & FutureViolations	Pass	5.834
BSVTSF, FutureViolations & FirstViolation	Pass	4.180

From this we see that even the most simple implementation of a sorting of the collisions from the feasible paths with a fitness swap can improve the system. The optimal configuration for this situation was in the case of a non fitness modifying violation type sort in conjunction with FutureViolations however this is simply case specific and should not be considered superior in other cases without further testing.

Benchmark Z4

This benchmark further confirms the effectiveness of the collision avoidance system.

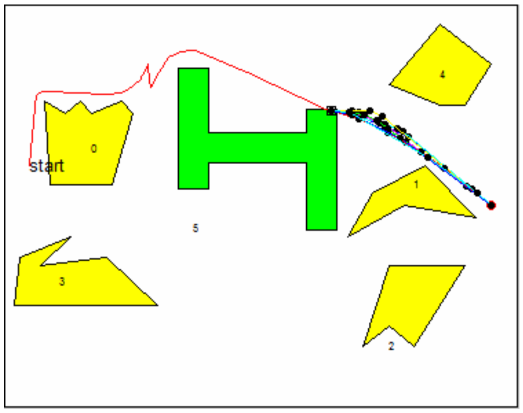


Figure 23: Base Case

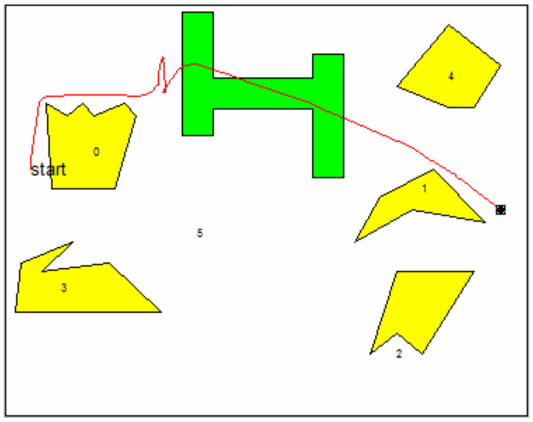


Figure 24: Collision Avoidance

Benchmark Z3

The z3 benchmark demonstrates the effect of the collision avoidance system has entered a seemingly unavoidable collision situation. It is shown that the path diversity of the collision avoidance system is greater and is working to reduce the effects of the collision by attempting to generate paths along the lower wall.

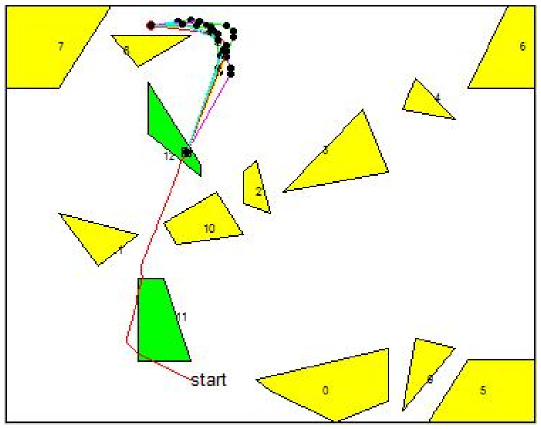


Figure 25: Base Case

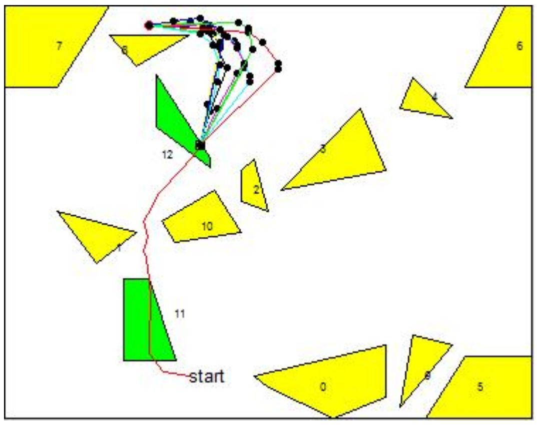


Figure 26: Collision Avoidance

Other cases all run into similar problems.

In all situations the collision scaling function was deemed ineffective despite proof that it was in fact working as planned. This is due to the operation of the elite paths function which chooses the first half of the list of paths to work with. The violation type bubble sort with fitness swapping enabled (which essentially performs the same operation as collision scaling while sorting the feasible paths to the front of the list) was much more effective as is shown in the Benchmark z1 table.

As shown in the z3 benchmark, the system is shown to be working, however in this case as with many others, other factors play a key role in the failure to escape collision. The best solution for the z3 scenario would be for the robot to pull away from the target and go under the moving obstacle. However, the likeliness of a path mutating to this solution is highly unlikely but not impossible. If the mutation rate was dynamically altered in these situations it could be possible to avoid collision. A solution that was discovered to work in many instances is a slight speed alteration. The following image depicts the z3 bench mark with a speed of 0.7 instead of 0.5.

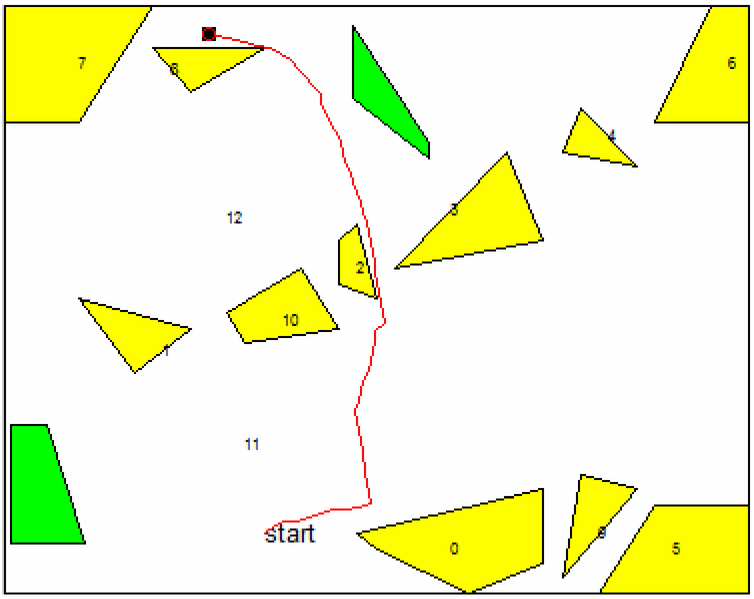


Figure 27: Effect of Speed Increase

Further testing reviled that mutation plays a key role in collision avoidance.

The following 2 figures depict the effect of increasing the mutation rate of the GA. The solution passes the first collision but fails the second collision. In Figure 29, the mutation rate increase and speed increase work together to attain the goal.

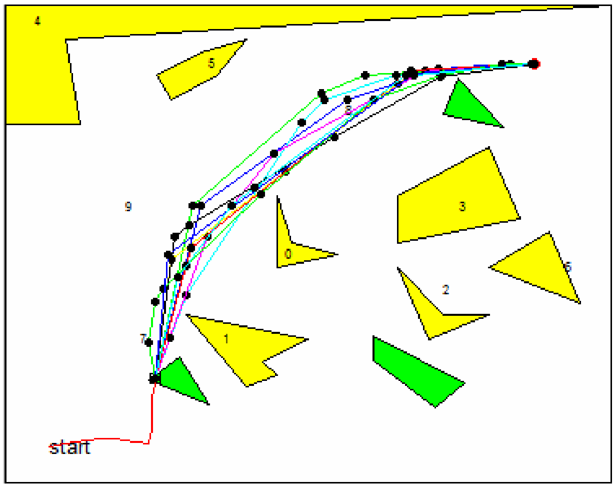


Figure 28: Lower Mutation Rate

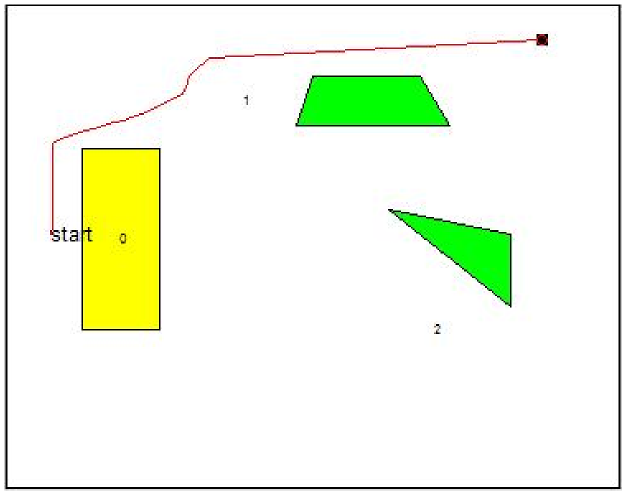


Figure 29: Higher Mutation Rate

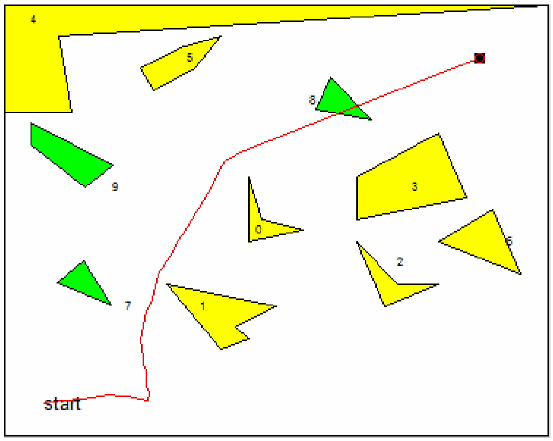


Figure 30: Higher Mutation Rate & Speed

It is our belief that a dynamic robot velocity with bounds and a dynamic mutation rate in tight situations could improve collision avoidance while maintaining realistic simulation.

4.2 Radius of Vision

Testing

There were several different types of tests run to determine if radial vision improved the functionality; to find the optimum number of nodes; to determine how using a percentage of nodes in the path affected the outcome; and finally to compare these three methods. For testing purposes the test file DynamismGA was used as seen in the appendix section of this report. Seven benchmark files were created for testing purposes and are also included in the appendix section.

4.2.1 Preliminary Testing

Initially it was necessary to determine if the concept of radial vision would have an impact on the program. If so, the nature of this impact could then be explored further. After several runs on benchmark R1 it was evident that the newly implemented vision system had an impact. Surprisingly, the robot crashed when run with normal vision and was successful with the radial vision. The following two figures demonstrate the results of the two runs.

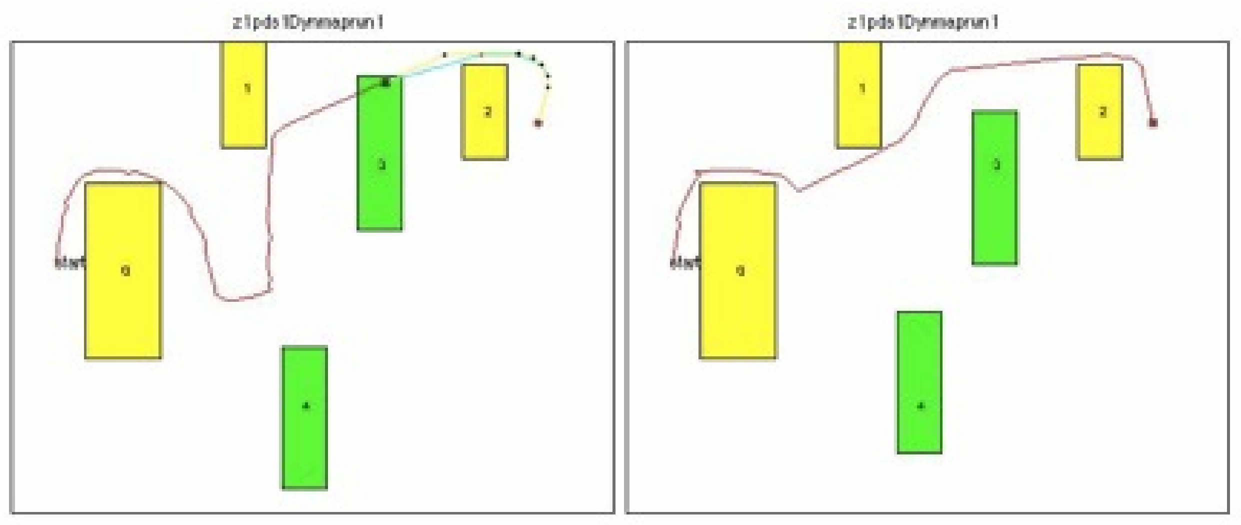


Figure 31 – Benchmark R1 Result for Normal Vision (left) & Radial Vision (right)

However, radial vision was only successful when run with the right number of nodes.

4.2.2 Optimal Number of Nodes Determination

Therefore the next logical step was to sequentially run radial vision with varying numbers of nodes for a range of benchmarks to determine if it was possible to calculate an optimum number that would provide a high success rate. Success is defined by efficiently completing a map with no violations. For each of the seven benchmarks the program was executed with varying node values ranging from five to thirty. The number of generations to reach a solution was recorded for each run. The intentions of this testing was to determine if a correlation exists between number of nodes and success rate. The results of these tests are included in spreadsheet format in the appendix. A summary of these results can be seen in the table below.

Benchmark	**Difficulty**	**Path Vision**	**Radial Vision Number Providing Success**
R1	5	NO	6, 11, 13, 14, 15, 18, 20, 21, 22-30
R2	4	YES	5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 20, 21, 22, 23, 24, 25
R3	6	NO	11, 12, 15, 16, 19, 21, 24, 25, 26, 27, 28, 29 30
R4	1	YES	5, 6, 7, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 28, 29, 30
R5	7	NO	6, 7, 9, 12, 13, 16
R6	3	YES	5, 6, 8, 12, 14, 15, 18, 19, 21-30
R7	2	YES	5, 6, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 26-30
****Note: Difficulty rating based on scale (easisest)1-5 (most difficult)**			

Although the success rate was high for each of the benchmarks, there is no correlation or pattern between radial vision number and; number of iterations to reach the target; number of static and dynamic obstacles. The figure below demonstrates the fact that there is no consistent relationship between the number of nodes and the number of iterations taken to reach a solution.

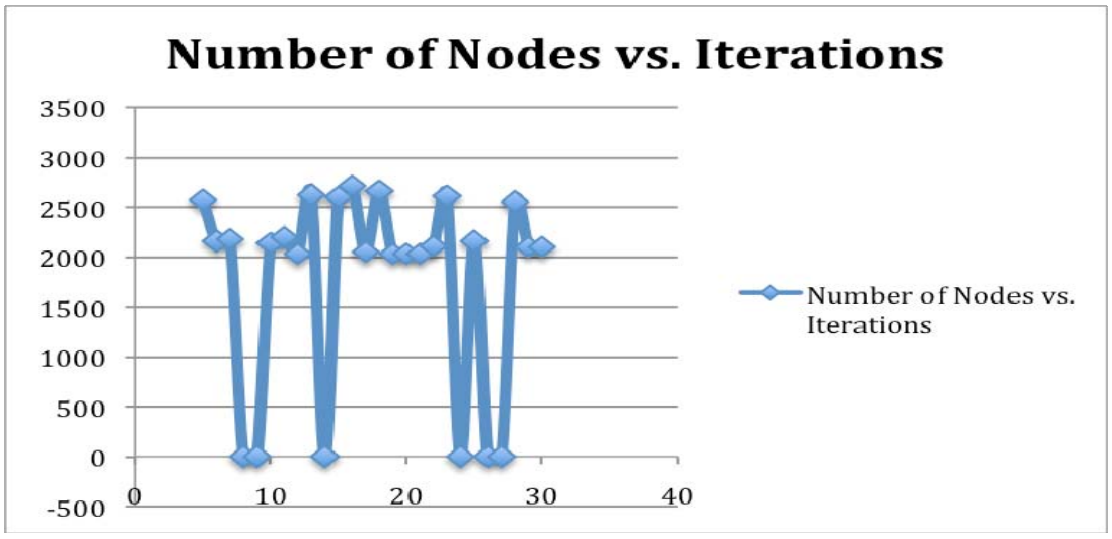


Figure 32 - Relationship Between Nodes & Iterations

There does however appear to be a relationship between Path Vision success and the success rate of radial vision. Looking at table BLANK seen above, the number of successful runs is greatly increased in the cases that Path Vision is successful. Also, in the cases where Path Vision is successful, radial vision optimizes the path taking fewer iterations to reach the target. The figure below supports this claim.

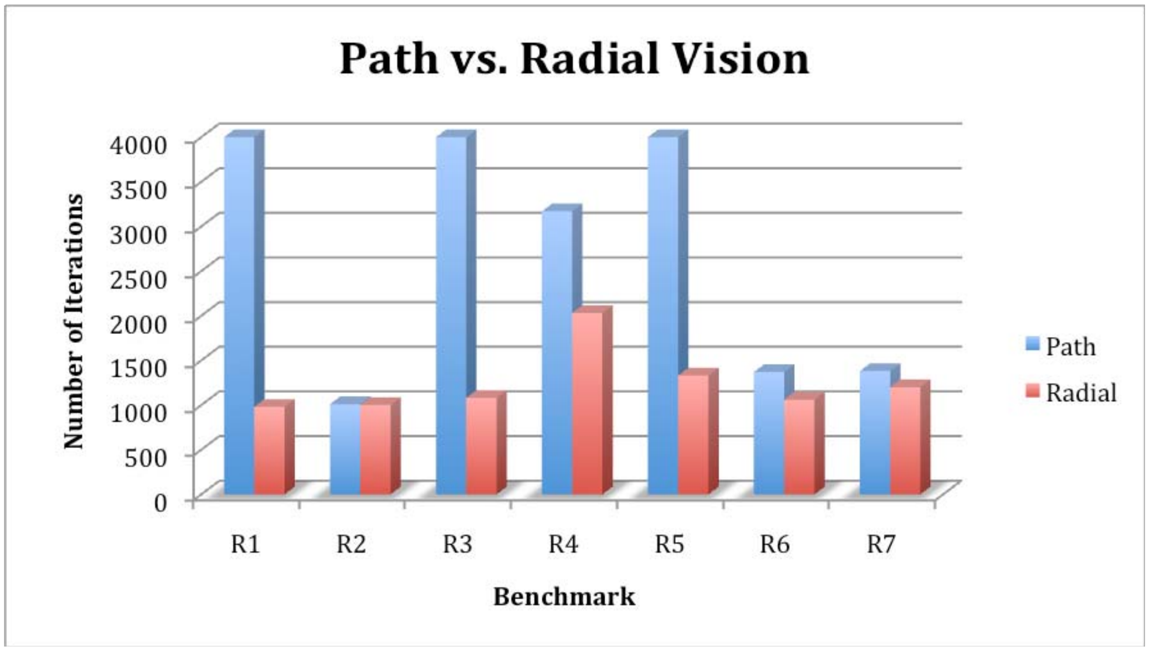


Figure 33 - Comparison of Path & Radial Vision Iterations

In the figure above, the benchmarks that are displayed as max number of iterations, a solution was either not reached or the robot crashed on route to the destination. These results suggests that the radial vision method of implementation is maybe not as optimal as it could be, or that if calculating an optimal number of nodes is not possible, it should be used more for optimizing an already successful path and not for collision avoidance.

Therefore, the modified radial vision was introduced. As explained earlier in this report, instead of evaluating the paths at a fixed number of nodes, they are evaluated by percentage of total nodes in the path. This would take into account that each path in the population has a varying number of nodes. Evaluating a percentage of the path would give the system more of an even radius. That is, with each iteration every path would be evaluated at approximately the same distance ahead of the robots current position. The modified system testing (included in the appendix) yielded a 33 percent success rate among benchmarks that failed with Path Vision, and a 51 percent pass rate in the cases where it was successful. These results showed that the original method of passing a fixed number of nodes was the better of the two implementations. Below is the same as figure 5 only the percentages were used instead of a fixed number of nodes.

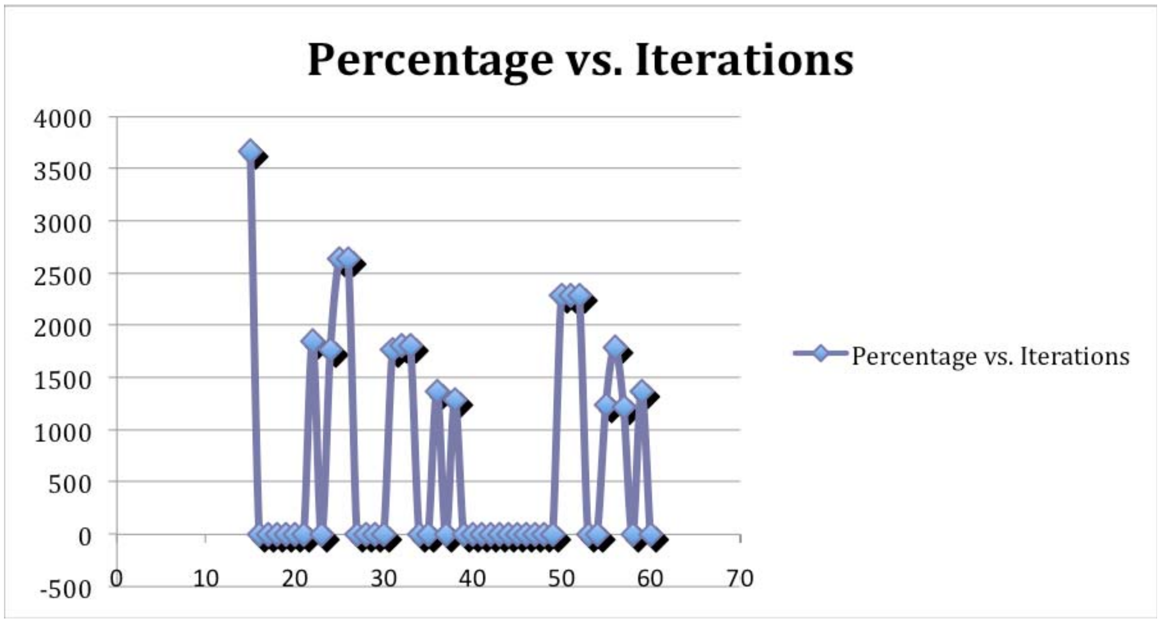


Figure 34 - Relationship Between Path Evaluation Percentage & Number of Iterations

In comparison with figure 5, the success rate among the path percentage evaluation tests is far lower than that of fixed nodes.

The range in values for testing these implementations was chosen based on consistency. When the results became consistently the same or very close in nature, that became the maximum for testing. The fact that there is a point where the numbers become very close is expected. As the fixed number of nodes or percentage increases to a certain point, the majority of the path will be evaluated at each iteration, and so Path Vision and Radial Vision become virtually the same.

4.2.3 Performance

The idea of not evaluating the entire path at each iteration is another attractive attribute to radial vision. Saving the time and resources in turn speeds up the processing time taken to evaluate the paths. However, the extra step of creating a new radial path at each iteration takes up a lot of resources. Below is a figure comparing the best and worst successful radial path trips with the successful path vision benchmarks.

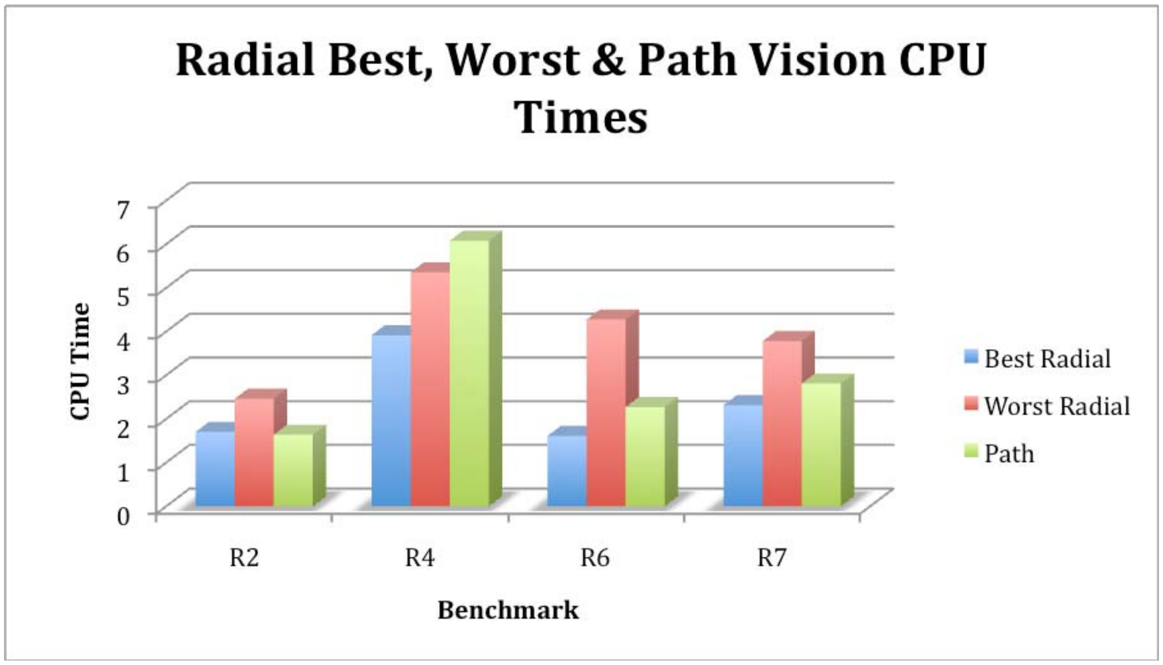


Figure 35 - Radial Best, Worst and Path Vision CPU Times

It appears that benchmark R4 created the scenario causing the greatest difference in CPU time. All other benchmarks yielded slightly better radial best CPU times. The radial worst CPU times were on average considerably worse than path vision times. The fact that the R4 benchmark takes considerably less CPU time suggests that in cases where the robot must travel great distances, radial vision is more efficient than path vision. These results reaffirm the fact that radial vision is a good solution optimizer when implemented properly. Even though there are cases where radial vision is successful and path vision is not, the radial implementation isn’t consistent enough to be used as collision avoidance.

4.3 PSO

4.3.1 Overview

In order to get an idea of the PSAP's performance in a static environment it was compared against the GAP in threee maps of varying complexity. Results were obtained for each algorithm solving each map for 100 iterations. The maps contained only static obstacles and robot velocity was set extremely low (~0.0001) so that its movement was minimal, and the effect negligable. Results were gathered as population best-so-far fitness values (averaged over 20 runs), and average population fitness values (also averaged over 20 runs). The results from each algorithm were compared, and this comparison was analyzed and is discussed later in this report. Results are displayed as charts (two charts for each map, with one chart showing best-so-far fitness values and the other showing average population fitness values) with results from the GAP and PSAP plotted on the same chart. The charts make clear the rates of convergance in each algorithm and also the population diversity. Paramaters for the PSAP were only mildly tuned and paramaters for the GAP were left at default settings

4.3.2 The Benchmarks

Three maps were used as benchmarks to compare the GAP to the PSAP and they are displayed in figures below. The run paramater and map paramater configurations can be found in the appendix.

Benchmark 1:

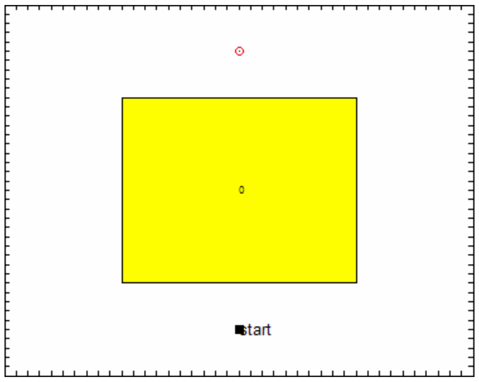


Figure 36 - Benchmark 1

Benchmark 2:

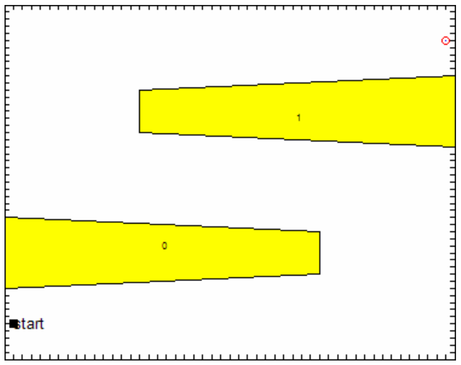


Figure 37 - Benchmark 2

Benchmark 3:

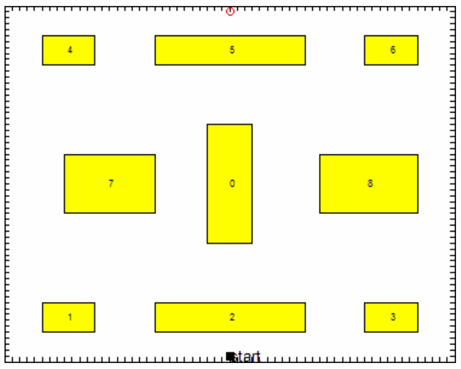
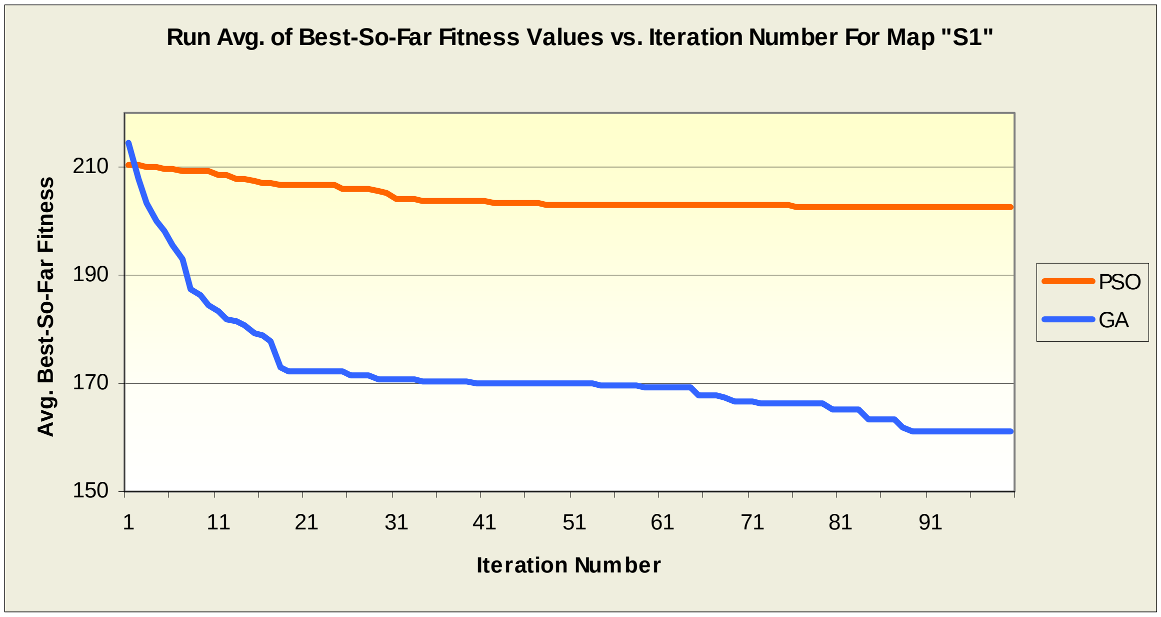


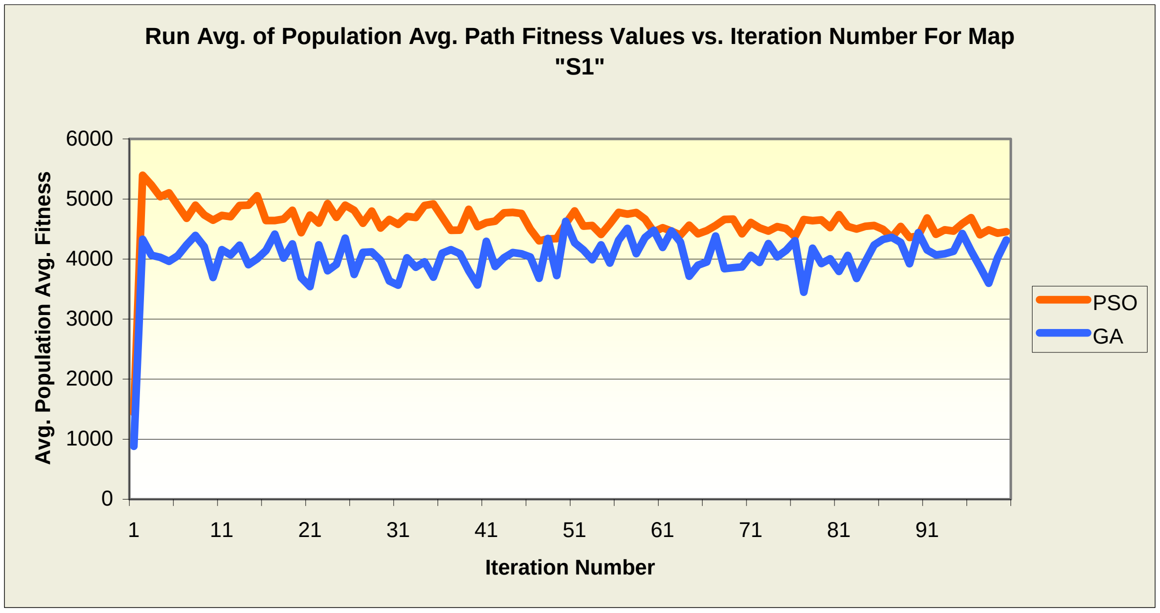
Figure 38 - Benchmar 3

4.3.3 Comparison Results

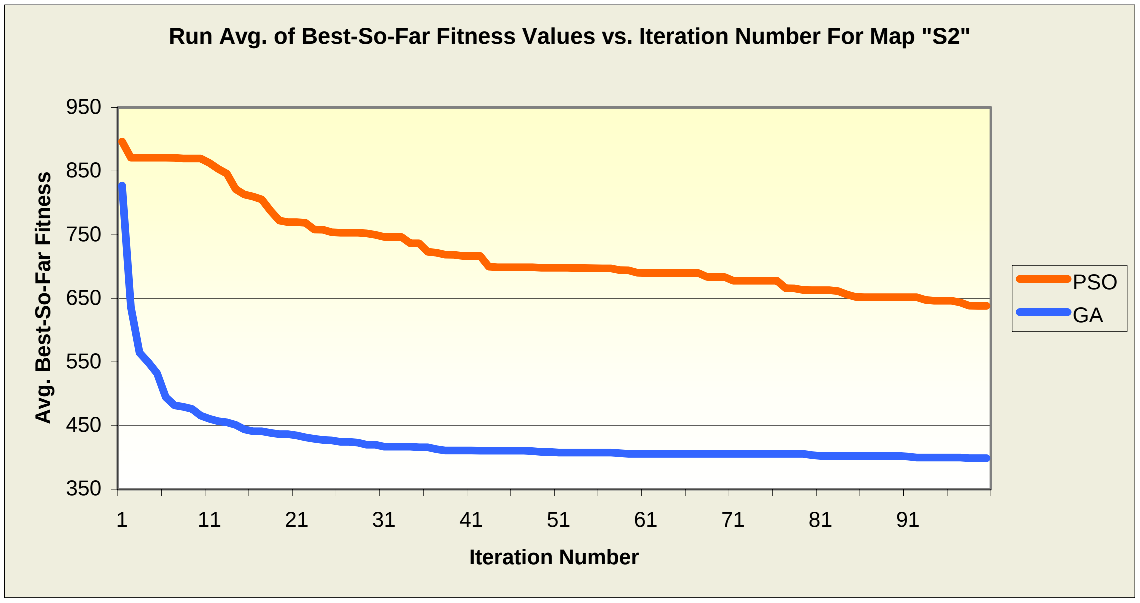
Results are divided by type and benchmark, with benchmark 1 results shown first, followed by benchmark 2 and then benchmark 3. Population best-so-far fitness values results are shown first for each benchmark followed by population average path fitness values results. The population best-so-far results are generated by taking the best path fitness value at each iteration for each run, and computing the average (across all runs) resulting best-so-far fitness at each iteration. Population average path fitness results are generated by computing the average path fitness for the current population at each iteration, for each run. The average (across all runs) for each iteration is computed.

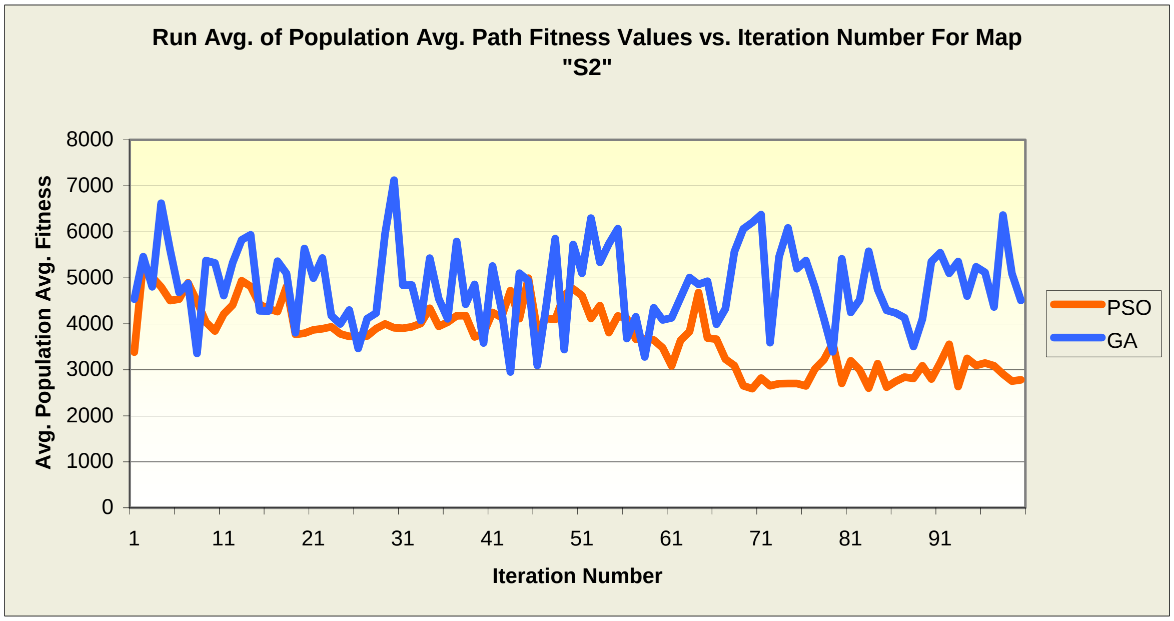
Benchmark 1:


Run Avg. of Best-So-Far Fitness Values vs. Iteration Number For Map "S1"
Avg. Best-So-Far Fitness
210
190
170
150
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA

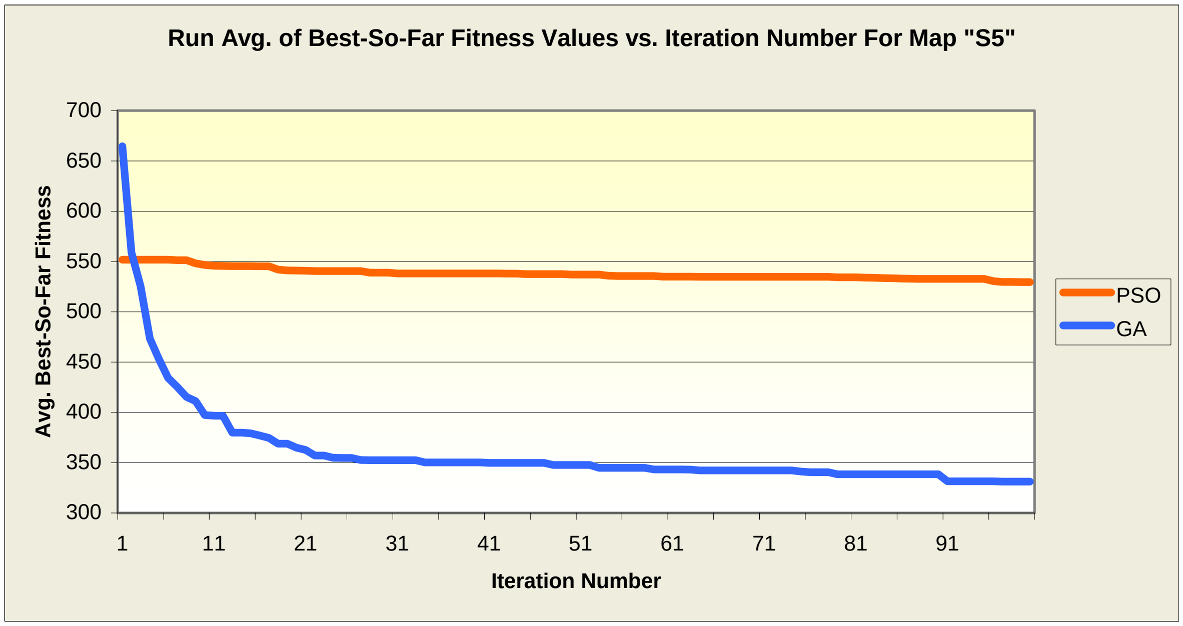

Run Avg. of Population Avg. Path Fitness Values vs. Iteration Number For Map "S1"
Avg. Population Avg. Fitness
6000
5000
4000
3000
2000
1000
0
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA

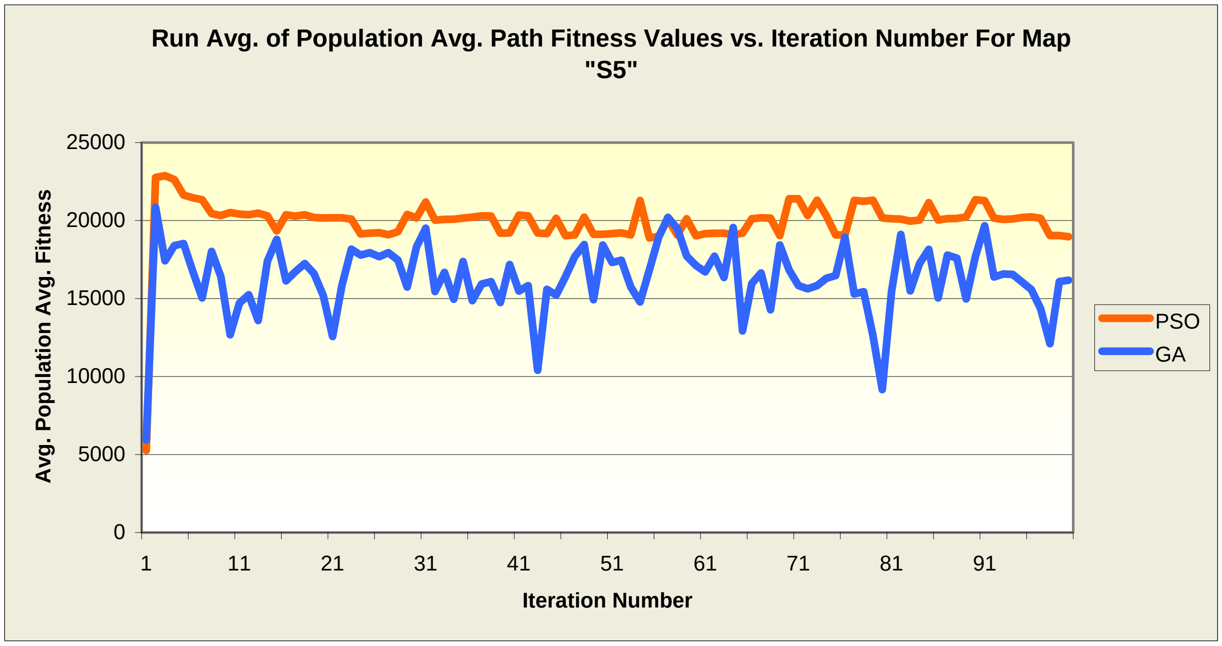
Benchmark 2:


Run Avg. of Best-So-Far Fitness Values vs. Iteration Number For Map "S2"
Avg. Best-So-Far Fitness
950
850
750
650
550
450
350
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA


Run Avg. of Population Avg. Path Fitness Values vs. Iteration Number For Map "S2"
Avg. Population Avg. Fitness
8000
7000
6000
5000
4000
3000
2000
1000
0
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA

Benchmark 3:


Run Avg. of Best-So-Far Fitness Values vs. Iteration Number For Map "S5"
Avg. Best-So-Far Fitness
700
650
600
550
500
450
400
350
300
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA


Run Avg. of Population Avg. Path Fitness Values vs. Iteration Number For Map "S5"
Avg. Population Avg. Fitness
25000
20000
15000
10000
5000
0
1
11
21
31
41
51
61
71
81
91
Iteration Number
PSO
GA

4.3.4 Comments

As can be seen by the best-so-far fitness values in each benchmark, the PSAP has a much slower convergence towards good solutions than the GAP. Also, the best path fitness values achieved by the PSAP are quite inferior to those attained by the GAP. It can be seen that as benchmark complexity increases the PSAP's final best-so-far fitness values trail the GAP's best-so-far fitness value by a growing amount. In benchmark 1 the PSAP's final best-so-far fitness is approximately 1.25 times as large as the GAP's, while in benchmark 2 it grows to about 1.55 times, and finally in benchmark 3 it is approximately 1.65 times as large. The PSAP's slow convergence may be a result of one or many of the drawbacks discussed in the Design section of this report.

The PSAP's average population fitness was similar to the GAP's but less noisy (changed less from iteration to iteration). This could be an indication that velocity and inertia paramaters were not set high enough in the PSAP, or may have been caused by many of the paths becoming infeasible (in which case the infeasible paths have very similar fitness values that are only somewhat worse than the feasible path fitness values). This suggests that further paramater tuning should be done and also that the PSAP's fitness evaluation function should be different than that of the GAP's. If often many solutions are indeed infeasible during the PSAP's execution then the average population fitness is not a good indicator of population diversity. It may be helpful to in the future create a method of accurately representing the populatiopn diversity.

Although inspection of the result's clearly shows PSAP's inferior performance to the GAP in solving a static environment, it is both unknown what the PSAP's performance would be like in a dynamic environment, and also what its performance would be like if it was further refined and improved. It is possible that some of the drawbacks in the design (such as not using a repair operator each iteration, using the GAP's fitness evaluation function, using a constant number of nodes, and allowing nodes to move to infeasible locations) are the root cause of the PSAP's poor results, and that the algorithm could indeed perform on a comparable level to the GAP.

5 Conclusions & Recommendations

The developments with the collision avoidance system reveal the complexities surrounding this issue. In some cases the system successfully guides the robot down safe paths however path diversity is needed in order for the collision avoidance system to be effective and controlled velocity of the robot will ultimately lead to better collision avoidance success.

It was shown that the collision avoidance system works as expected however due to design limitations the effectiveness of the system is not as strong as was originally expected. Future work should focus on developing dynamic velocity for the robot as well as dynamic mutation rate.

There is no question that radial vision is a concept worth implementing. Although it yielded some interesting results and provided a glimpse of a possible collision avoidance solution, its currently best application is optimization. Upon further development there might be more uses for this strategy. Current recommendations for further development would be to determine some kind of relation between factors like path length, number of obstacles, map complexity and optimal number of nodes. If its possible to derive a formula to determine the optimal number of nodes for a given path then there is a great possibility that radial vision can be used as a collision avoidance method.   Further investigation as to why evaluating a percentage of a path yielded such terrible result may be worth looking into. Radial vision is an interesting concept that makes sense to implement. Upon further research and testing, the current implementation can be refined.

The PSO algorithm implementation was successful in solving static environments but its performance was inferior to that of its GA counterpart. Much work can be done to improve the

PSAP and it is unknown how strongly these improvements will improve the PSAP's performance. It would be interesting and useful to see how the PSAP performs in a dynamic environment compared to the GAP. Future improvement of the PSAP and further investigation into its performance is encouraged, especially in a dynamic environment.

6 References

[1] Clerc M., Kennedy J., "The Particle Swarm-Explosion, Stability, and Convergence in a Multidimensional Somplex space", IEEE Transaction on Evolutionary Computation, 2002,vol. 6, p. 58-73.

[2] Xiaohui Hu, "Particle Swarm Optimization: Tutorial." Particle Swarm Optimization. 2006. 13 Oct 2008 <http://www.swarmintelligence.org/tutorials.php>.

[3] Ahmed Elshamli, "Mobile Robots Path Planning Optimization in Static and Dynamic Environments", Thesis Presented to The Faculty of Graduate Studies of University of Guelph, August, 2004.

7 Appendices

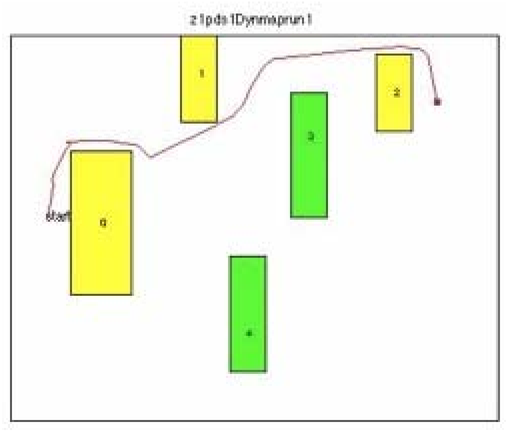


Figure 39 - R1 Benchmark

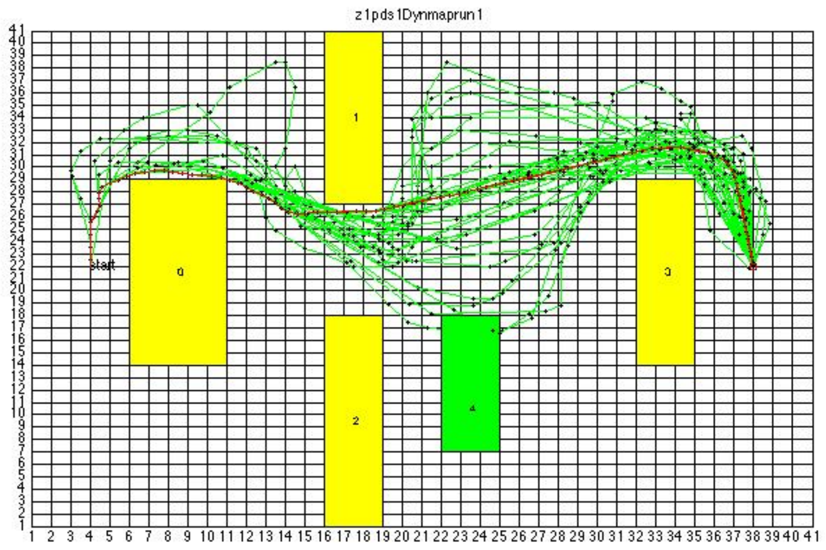


Figure 40 - R2 Benchmark

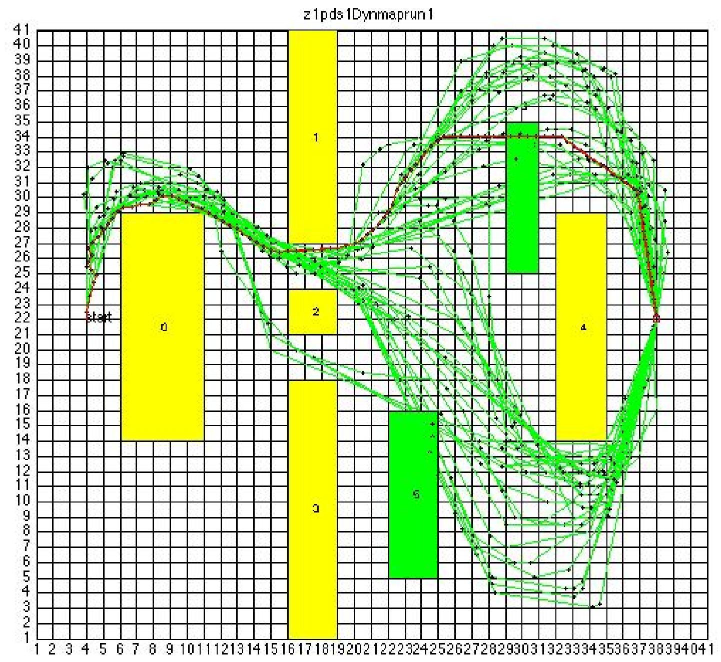


Figure 41 - R3 Benchmark

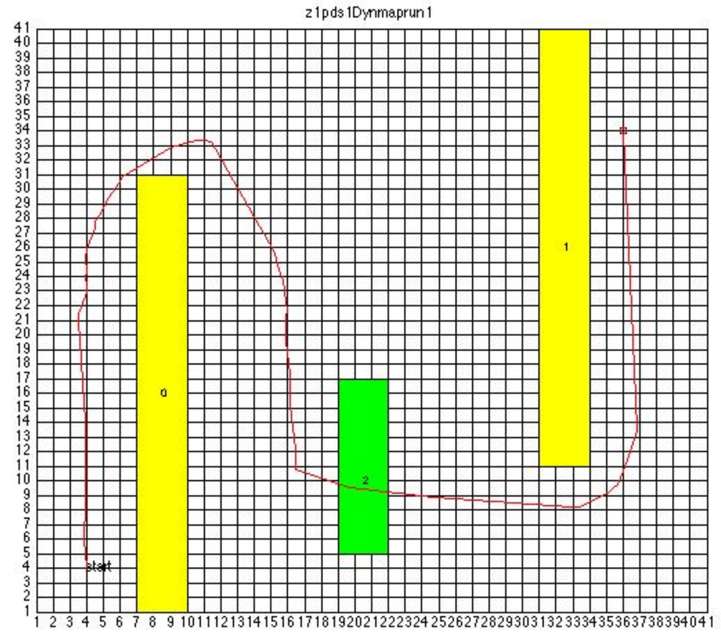


Figure 42 - R4 Benchmark

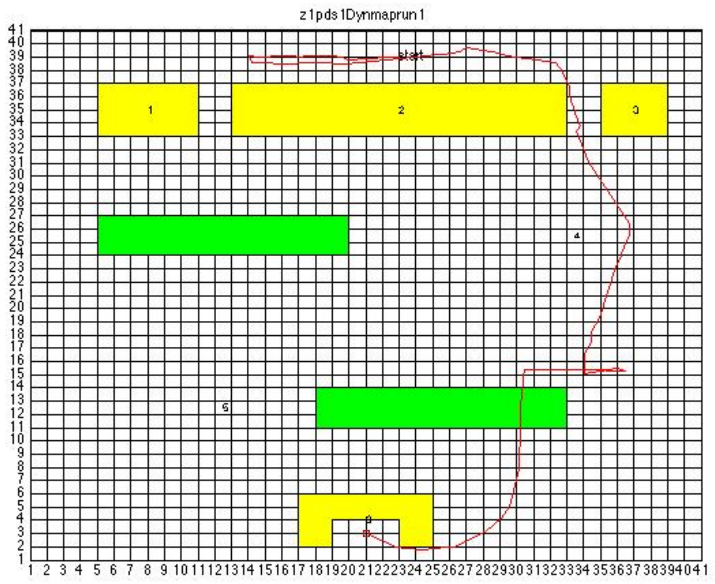


Figure 43 – R5 Benchmark

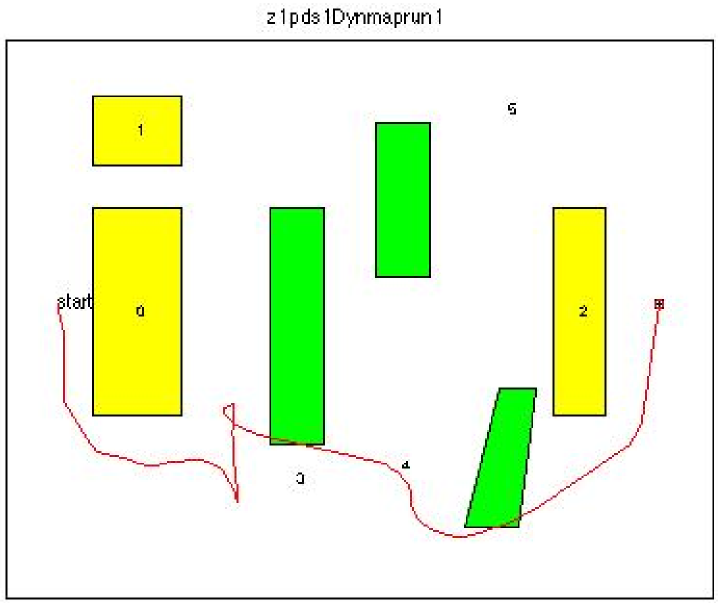


Figure 44 - R6 Benchmark

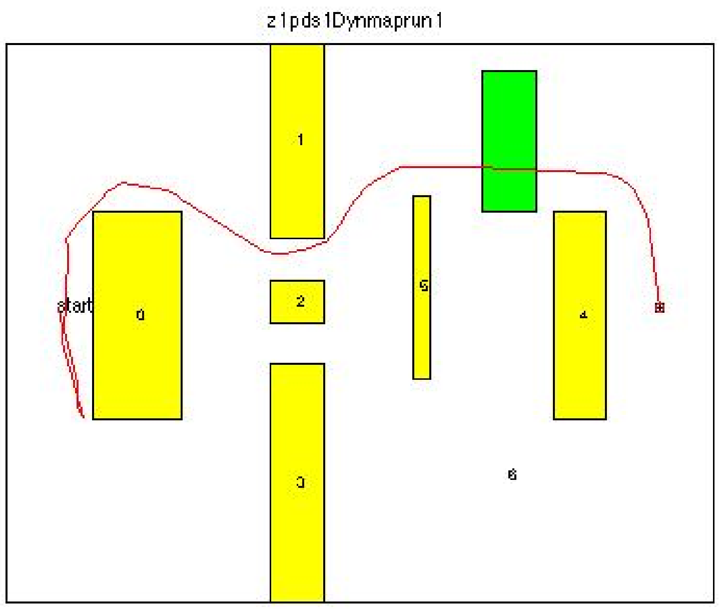


Figure 45 - R7 Benchmark

DynamismGA as used for Radial testing

```
./bin/dyn_path.exe +EfiR7.out +EfdT1c.txt +Efoz1Pds1 +Ern1 +Ppa1 +Gpn20 +Gmg4000 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5  +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0  +Pds1 +Pdf2 +Pdr10 +Pdv0.5 +Pdg10 +Pwa0 +Din5
```

7.1 Intelligent Collision Avoidance System (ICAS)

In order to address the problems and shortcomings of PCAS, a new solution is being developed. ICAS (Intelligent Collision Avoidance System) will address the major causes of problems and shortcomings present in PCAS by providing methods of adding new nodes to solutions when necessary, properly assessing clearance with moving obstacles, and by removing collision detection and collision cost and instead treating collisions in the same manner that infeasible solutions are currently detected and penalized. ICAS will accomplish these achievements by mapping moving obstacles into a time domain. In this way, the GAP will be able to see where the obstacles will be in the future and how the paths will relate to these obstacles. ICAS will allow the system to treat moving obstacles as static ones, but redraw them as each path would see them in the future. In this way, ICAS will allow the GAP to retain all of its currently tested and properly working operators to correctly and reliably add new nodes where needed and to correctly calculate the fitness function in a manner that causes predictable and desirable convergence. In this way, ICAS in its basic form will be able to outperform PCAS with solutions and improvements added. Also, ICAS will still allow for improvements such as adaptive parameters and a new mutation operator to be added later.

7.1.1 System Overview

ICAS attempts to see into the future in great detail and accuracy. It does so by predicting the approximate shape and position of an obstacle, as each path will encounter it as it passes through the obstacle's AOM (area of motion). The system starts by removing all moving obstacles from the standard methods of evaluation and operation. Moving obstacles are evaluated and operated on in the same way as non-moving obstacles but must be redrawn on a time-domain for each path individually before this can be done. ICAS next generates an AOM for each moving obstacle in the map. The AOM is the approximate area that the obstacle will pass through in a certain number of iterations in the future. It is approximate because it is based on the encompassing rectangle of an obstacle and not the actual obstacle itself. This approximation reduces program complexity and improves performance. An example of the AOM is displayed as the lightly shaded region in front of the obstacle in Figure 1.08 and Figure 1.09.

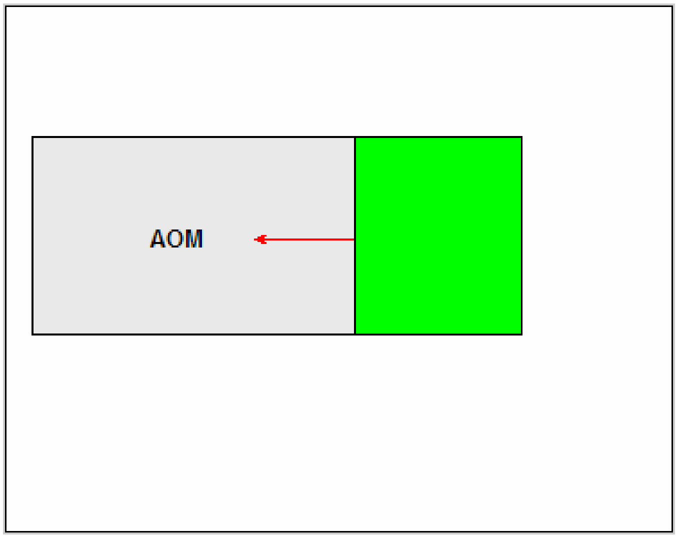


Figure 11– Area of Motion Horizontal Motion

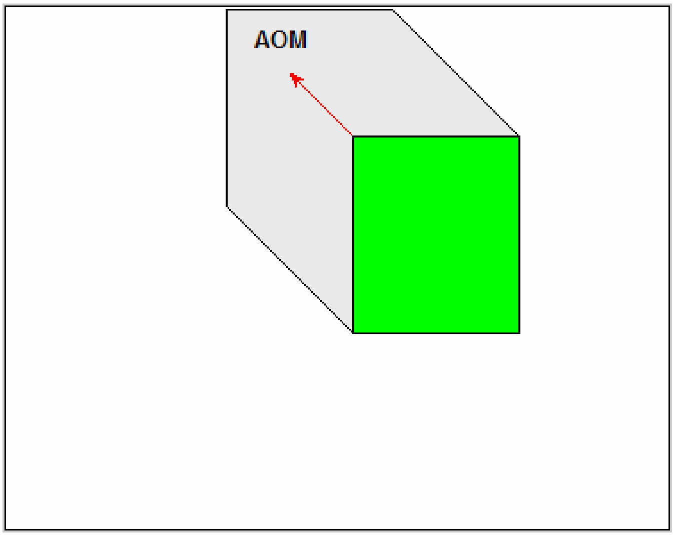


Figure 12- Area of Motion Angled Motion

The AOM is essentially a projection of the obstacle's encompassing rectangle along its vector of motion. The length of this projection is the product of the number of iterations that ICAS wishes to look into the future and the speed of the obstacle. ICAS uses the AOM to determine which paths will be evaluated according to the motion of the obstacle (ie, paths that lie outside of the AOM can be ignored to save processing power).

For each path that crosses a portion of the AOM, ICAS generates a new object unique to that path called the PDO (Path Dynamic Obstacle). The PDO is the obstacle that the GAP will use to evaluate the path's fitness and to perform its operators on (ie, repair). The path fitness as evaluated around the path's PDOs is then added to the fitness as evaluated around the static obstacles. When the repair operator is called for each path, it looks at both the static objects that the path crosses through and also the PDOs that each path crosses through. Repairs are made accordingly. The creation of each PDO is complex and is explained in the next section.

7.1.2 PDO Creation

Before a PDO can be created, ICAS first determines how a path is crossing the AOM in question. There are two base cases.

Case 1: The path crosses the AOM with a single line (there are no nodes that lie inside the AOM)

Case 2: The path crosses the AOM with multiple lines (there are one or more nodes inside the AOM)

These cases are displayed in Figure 1.10 and Figure 1.11. Each case 2 type of path crossing must be partitioned into as many case 1 type crossings as necessary. The fitness evaluation function and repair operator must be performed individually on each case 1 partition and the results combined. An example of this partitioning is shown in Figure 1.12.

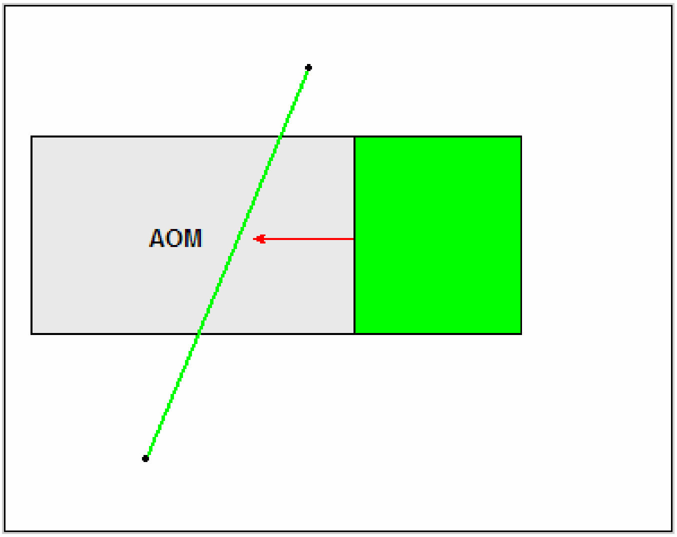


Figure 13 – Case 1 Path Crossing

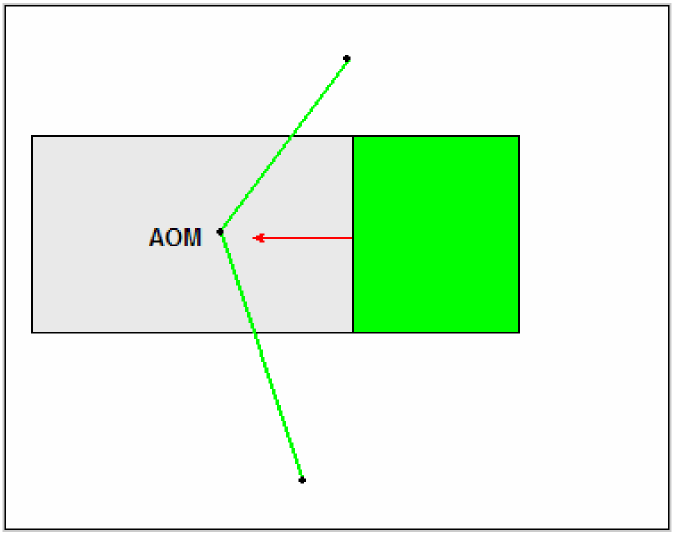


Figure 14 – Case 2 Path Crossing

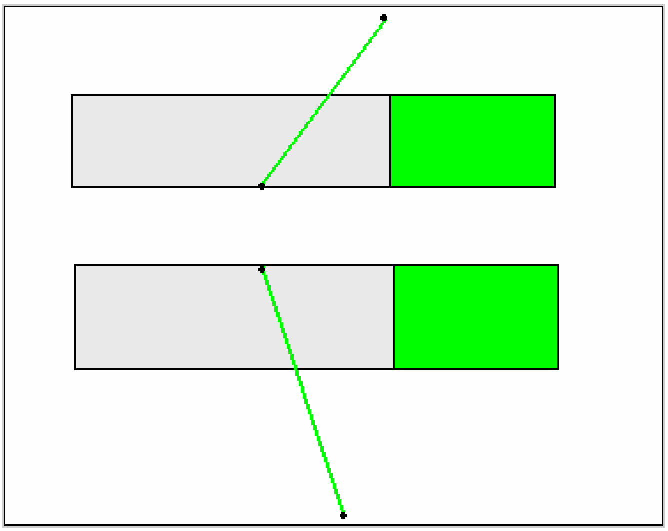


Figure 15 – Case 2 Partitioned

Each Case 1 path crossing can then be organized into one of three possible conditions:

Condition 1: Path enters AOM on one edge parallel to obstacle’s DOM (direction of motion) and leaves on opposing edge parallel to object’s DOM.

Condition 2: Path enters AOM on one edge perpendicular to obstacle’s DOM and leaves on one edge parallel to object’s DOM.

Condition 3: Path enters AOM on one edge perpendicular to object’s DOM and leaves on opposing edge parallel to object’s DOM.

These three possible conditions are displayed in Figures 1.13 through 1.15. Condition 2 and 3 path crossings are trimmed to become Condition 1 path crossings. This is shown in Figure 1.16.

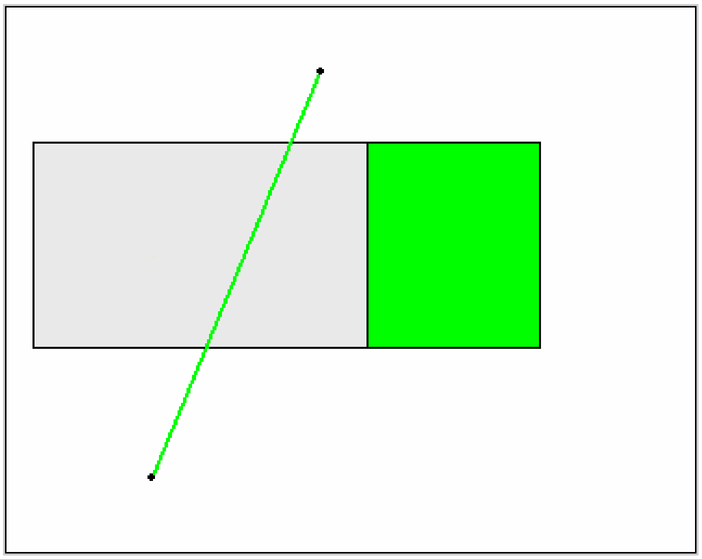


Figure 16 – Condition 1

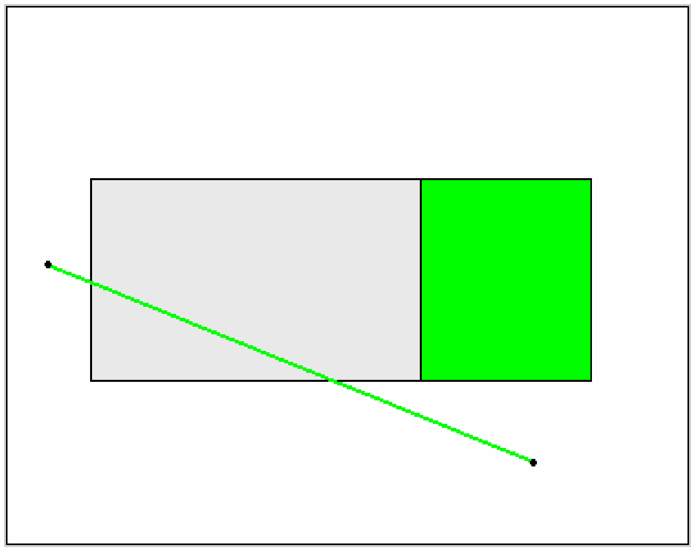


Figure 17 – Condition 2

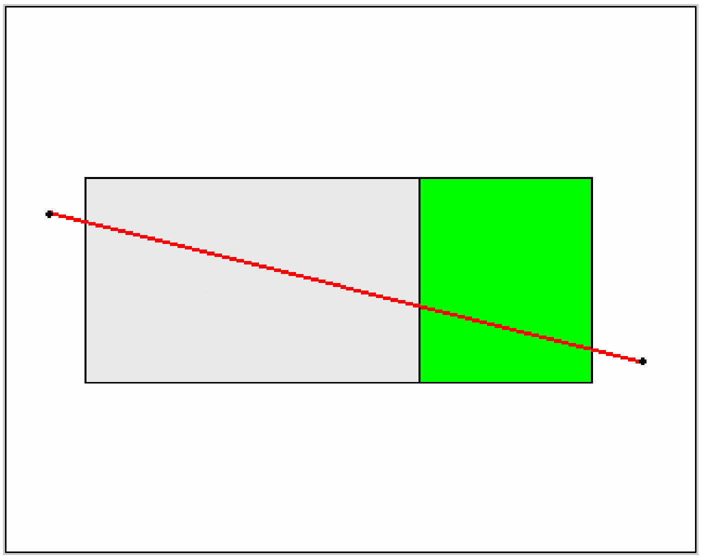


Figure 18 – Condition 3

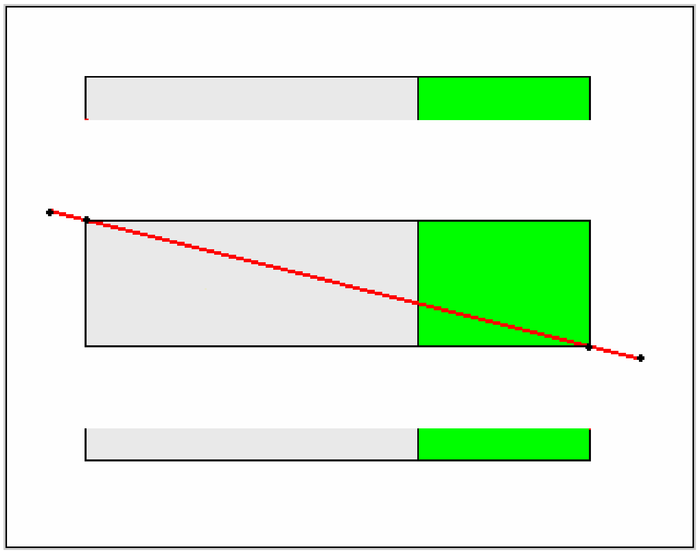


Figure 19 – Trimming to Condition 1

Next, ICAS will draw the PDO for each path at each AOM the path crosses. In this step, ICAS looks at each of the two edges where the path crosses the AOM. At each edge, ICAS determines at what iteration the robot will reach the path/edge intersection. ICAS then redraws the two obstacle vertices that are located on that edge, as they will appear at that iteration. The four newly created obstacle vertices are then joined to form the PDO. This process is illustrated in Figure 1.17.

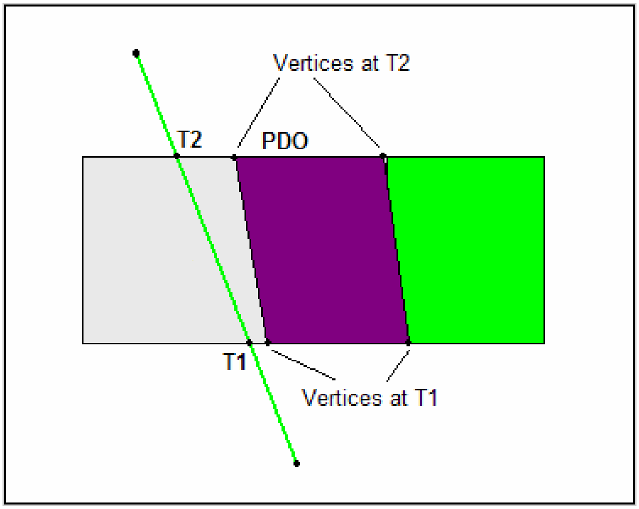


Figure 20 – PDO Creation

7.1.3 ICAS Benefits

The benefits of using ICAS as opposed to PCAS will be that ICAS will be able to quickly guide the GAP into regions of the solution space that not only avoid collision, but also avoid collisions in an optimum manner. ICAS overcomes the problem of generating new nodes that PCAS faces by allowing the GAP to use the repair function on the PDOs that ICAS creates. Also, ICAS allows the GAP to properly use its clearance cost factor in the fitness function which both speeds up convergence to good solutions and also allows for better solutions to be created. Rather than heavily modifying the existing fitness function and GAP operators, ICAS leaves the current system in tact and instead attempts to interface with the current system. By interfacing with existing operators and fitness function evaluation that has been extensively developed, if successfully implemented, ICAS will be a reliable and stable collision avoidance system that can later be expanded and improved on.

7.1.4 Possible Difficulties and Problems With ICAS

Although ICAS has many benefits over PCAS, there are several possible problems and difficulties with the system. One difficulty will be apparent when dealing with partitioning Case 2 path crossings. Each partition must be evaluated separately by the fitness function and operated on separately by the repair operator. These individual operations must then be combined back into a whole such that they work with the whole solution. It may be very difficult to find a method of performing repair operators such that they do not conflict with each other or form poor solutions. One possible solution is to allow the repair operator to function only once per each set of Case 1 partitions (partitions that came from a common Case 2 path crossing). It is unknown how well such a solution would perform. Another problem that may arise is computational difficulty. Creating a PDO for each path at each AOM that it encounters may prove to be computationally challenging and time consuming. A possible solution to this may be to only generate a PDO for the first AOM that each path crosses, and simply ignore any other AOMs the path may cross. It is unknown how such a solution may perform.

7.2 Code

7.2.1 Collision Avoidance

Collision Scaling

```
void colision_scalling(){
    double ncwf = 0.0; //no colision worst fitness
    double dcbf = MAX_FLOAT; //dynamic colison best fitness
    double dcwf = 0.0; //dynamic colison worst fitness
    double scbf = MAX_FLOAT; //static colison best fitness
    int i = 0;

    for (i=0;i<PopSize;i++){
        if (PathsPop[i].ViolationType == 0){
            if (PathsPop[i].fittness > ncwf){
                ncwf = PathsPop[i].fittness;
            }
        }
        if (PathsPop[i].ViolationType == 1){
            if (PathsPop[i].fittness < dcbf){
                dcbf = PathsPop[i].fittness;
            }
        }
    }
//scale dc
    if (dcbf < ncwf){
      for (i=0;i<PopSize;i++){
        if (PathsPop[i].ViolationType == 1){
        PathsPop[i].fittness = PathsPop[i].fittness + (ncwf - dcbf) + 1;
        }
      }
    }

//get dcwf scbf
    for (i=0;i<PopSize;i++){
        if (PathsPop[i].ViolationType == 1){
            if (PathsPop[i].fittness > dcwf){
                dcwf = PathsPop[i].fittness;
            }
        }
        if (PathsPop[i].ViolationType == 2){
            if (PathsPop[i].fittness < scbf){
                scbf = PathsPop[i].fittness;
            }
        }
    }

//scale sc
    if (scbf < dcwf){
      for (i=0;i<PopSize;i++){
        if (PathsPop[i].ViolationType == 2){
        PathsPop[i].fittness = PathsPop[i].fittness + (dcwf - scbf) + 1;
        }
      }
    }

}
```

Obtaining Time to Collision and Type of Collision

```
void get_future_violations(O_STRUCT * map){

   int gens_ahead; //how many generations ahead we are looking
   int generation;
   int i, j, k, isEnvChanged, v, isFirstViolation;
   double del_x, del_y;
   POINT botposition=start_p;
   O_STRUCT * temp_map=NULL;
   PATH_STRUCT temp_path;
   NODE * current;

   for (k=0; k<PopSize; k++){
      PathsPop[k].FutureViolations = 0;
      isFirstViolation = 0;
      temp_map=NULL;
      generate_all_shadows(2, &temp_map);

      //duplicate the path
      temp_path.head=NULL; temp_path.tail=NULL; temp_path.node_num=0;
      current = PathsPop[k].head;
      while (current!=NULL){
         add_node_to_end(&temp_path, current->x, current->y);
         current=current->next;
      }

      //go forward in time
      PathsPop[k].ViolationType = 0;
      for (gens_ahead=0; gens_ahead<500; gens_ahead++){
         generation = cur_gen + gens_ahead;

         //move robot
         if (generation % num_generations_between_bot_move == 0){
            isEnvChanged=1;
            botposition = move_dist_on_path(bot_Velocity,&temp_path, 0);
         }

         generation++;

         //move obstacles
         if (generation % dyn_rate == 0){
            isEnvChanged=1;
            for (i = Stationary_obs_count; i< Initial_obs_count ;i++){
            // i.e. only moving obstacles
               if (temp_map[i].o_type == 2)
               {

                  del_x = dyn_severity * temp_map[i].move_sense *
                          temp_map[i].lambda_x;
                  del_y = dyn_severity * temp_map[i].move_sense *
                          temp_map[i].lambda_y;
                  for (j=0;j<temp_map[i].v_count;j++){
                     temp_map[i].vertcs[j].x += del_x;
                     temp_map[i].vertcs[j].y += del_y;
                  }
               }
            }
         }

         if(isEnvChanged==1){
            isEnvChanged=0;
            //check for a collision
            v = check_collision_and_type(temp_map,&botposition);
            if (v != 0){
               PathsPop[k].FutureViolations =
                   PathsPop[k].FutureViolations+1;
               if (isFirstViolation == 0){
                  isFirstViolation = 1;
                  PathsPop[i].FirstViolation = generation;

               }
            }
            if ((v == 1) && (PathsPop[k].ViolationType != 2)){
            //dynamic obstical colision
               PathsPop[k].ViolationType = 1;
            }
            if (v == 2){ //static obstical colision
               PathsPop[k].ViolationType = 2;
            }
         }
      }
   } //end for loop, next path
   free(temp_map);
   delete_list(&(temp_path.head));
}

int check_collision_and_type(O_STRUCT * map, POINT * p){
   int i;
   O_STRUCT obst;
   //loop through obstacles
   for (i = 0; i< obstacles_count ;i++){
      obst = My_Map[i];
      //are we colliding with this obstacle?
      if (is_pointInPolygon(&obst, (POINT*)
         obst.vertcs,obst.v_count,*p)==TRUE){
         if ((obst.o_type==2)||(obst.o_type == 1)||(obst.o_type == 11)){
            return 1;
         }
         if (obst.o_type == 0){
            return 2;
         }
      } else {
         //printf("No Colision \n");
      }
   }
   return 0;
}
```

First Violation & Future Violation Scaling

```
void buble_sort_ViolationType(){
   int done = 0;
   int i = 0;
   double temp_fitness;
```

```
      PATH_STRUCT temp_path;
      while(!done){
         done = 1;
         for (i=0;i<(PopSize-1);i++){
            if (PathsPop[i].ViolationType > PathsPop[i+1].ViolationType){
               done = 0;
               if (PathsPop[i].fittness > PathsPop[i+1].fittness){
                  temp_fitness = PathsPop[i].fittness;
                  PathsPop[i].fittness = PathsPop[i+1].fittness;
                  PathsPop[i+1].fittness = temp_fitness;
               }
               temp_path = PathsPop[i];
               PathsPop[i] = PathsPop[i+1];
               PathsPop[i+1] = temp_path;
            }
         }

      }
   }

   void buble_sort_FirstViolation(){
      int done = 0;
      int i = 0;
      double temp_fitness;
      PATH_STRUCT temp_path;
      while(!done){
         done = 1;
         for (i=0;i<(PopSize-1);i++){
            if ((PathsPop[i].ViolationType>0)
            &&(PathsPop[i+1].ViolationType>0)){
               if ((PathsPop[i].ViolationType ==
               PathsPop[i+1].ViolationType)&&(PathsPop[i].FirstViolation <
               PathsPop[i+1].FirstViolation)){
                  if (PathsPop[i].fittness < PathsPop[i+1].fittness){
                     done = 0;
                     temp_fitness = PathsPop[i].fittness;
                     PathsPop[i].fittness = PathsPop[i+1].fittness;
                     PathsPop[i+1].fittness = temp_fitness;
                     temp_path = PathsPop[i];
                     PathsPop[i] = PathsPop[i+1];
                     PathsPop[i+1] = temp_path;
                  }
               }
            }
         }
      }
   }

   void buble_sort_FutureViolations(){
      int done = 0;
      int i = 0;
      double temp_fitness;
      PATH_STRUCT temp_path;

      while(!done){
         done = 1;
         for (i=0;i<(PopSize-1);i++){
            if ((PathsPop[i].ViolationType>0)&&
            (PathsPop[i+1].ViolationType>0)){
               if ((PathsPop[i].ViolationType ==
               PathsPop[i+1].ViolationType)&&(PathsPop[i].FutureViolations
               < PathsPop[i+1].FutureViolations)){
                  if (PathsPop[i].fittness < PathsPop[i+1].fittness){
                     done = 0;
                     temp_fitness = PathsPop[i].fittness;
                     PathsPop[i].fittness = PathsPop[i+1].fittness;
                     PathsPop[i+1].fittness = temp_fitness;
                     temp_path = PathsPop[i];
                     PathsPop[i] = PathsPop[i+1];
                     PathsPop[i+1] = temp_path;
                  }
               }
            }
         }
      }
   }
```

#PARAMATERS USED FOR PSO RESULTS

#BENCHMARK 1 - PSO

```
./bin/dyn_path.exe +EfiS1.out +EfdT1c.txt +EfoS1_PSO +Ern20 +Ppa6 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0 +Spn8 +Sps1 +Spr0.1 +Svx2 +Svy2 +Svi0.8 +Sal0.8 +Sag0.5 +Pds1 +Pdf2 +Pdr1
+Pdv0.0001 +Pdg1 +Pwa0 +Din1
```

#BENCHMARK 1 - GA

```
./bin/dyn_path.exe +EfiS1.out +EfdT1c.txt +EfoS1_GA +Ern20 +Ppa1 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0 +Spn8 +Sps1 +Spr0.1 +Svx2 +Svy2 +Svi0.8 +Sal0.8 +Sag0.5 +Pds1 +Pdf2 +Pdr1
+Pdv0.0001 +Pdg1 +Pwa0 +Din1 +Ero1
```

#BENCHMARK 2 - PSO

```
./bin/dyn_path.exe +EfiS2.out +EfdT1c.txt +EfoS2_PSO +Ern20 +Ppa6 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0 +Spn10 +Sps1 +Spr0.2 +Svx2 +Svy2 +Svi1.4 +Sal0.8 +Sag0.3 +Pds1 +Pdf2 +Pdr1
+Pdv0.0001 +Pdg1 +Pwa0 +Din1 +Ero0
```

#BENCHMARK 2 - GA

```
./bin/dyn_path.exe +EfiS2.out +EfdT1c.txt +EfoS2_GA +Ern20 +Ppa1 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0 +Spn10 +Sps1 +Spr0.2 +Svx2 +Svy2 +Svi1 +Sal0.8 +Sag0.3 +Pds1 +Pdf2 +Pdr1
+Pdv0.0001 +Pdg1 +Pwa0 +Din1 +Ero1
```

```
#BENCHMARK 3 - PSO
```

```
./bin/dyn_path.exe +EfiS5.out +EfdT1c.txt +EfoS5_PSO +Ern20 +Ppa6 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20  +Gtg0
+Gir0 +Gim0 +Spn12 +Sps1 +Spr0.15 +Svx1 +Svy1 +Svi0.4 +Sal0.8 +Sag0.1 +Pds1 +Pdf2 +Pdr1
+Pdv0.0001 +Pdg1 +Pwa0 +Din1
```

```
#BENCHMARK 3 - GA
```

```
./bin/dyn_path.exe +EfiS5.out +EfdT1c.txt +EfoS5_GA +Ern20 +Ppa1 +Gpn20 +Gmg100 +Gif0
+Gcv2 +GRR10 +Gmt5 +Pwd1 +Pws5 +Pwc5 +Grx0.5 +Grm0.5 +Gri0.7 +Grs0.5 +Guc20
```

7.3 User Manual

List of Functions

```
void computePopFitness(PATH_STRUCT* t_paths,int nPths)
```

After initialization there are a list of paths. For each path it is determined if there are any violations. This function then takes the feasible paths and calculates the fitness of each one by calling the function calculateFeasiblePathFitness seen below. Affter a fitness is assigned to each path in the population, the function then evalutes each infeasible baded on the following.

```
t_paths[i].fittness = path. WorstFeasibleFitness +
                        t_paths[i].avg_infeasible +
                        t_paths[i].Violations;// was  t_paths[i].total_intersections;
```

```
double calculateFeasiblePathFitness(PATH_STRUCT* pth1)
```

This function is responsible for assigning a fitness to all of the feasible paths in the population. It receives one single path out of the population for evaluation and returns a fitness value. If the radial vision flag isn't set, then the fitness is based on the following:

```
pathCost = wd * pth1->length +  ws * pth1->smthCost  +  wc *  pth1    ->clrCost + wca * pth1->CollisionCost;
```

So basically each path is evaluated based on its length, smoothness, clearance, and collision costs.

If the radial flag is set, another function called createRadialPath() is then called to generate a new list based on a given radius.

```
int get_time_to_collision(O_STRUCT * map, PATH_STRUCT path)
```

This function basically looks at future iterations to determine how much time there is before a collision occurs.

```
void updateEliteSolutions()
```

This is an interesting function. Each time a new generation is generated, the best paths from both generations are put into an Elite solutions list. This gives the program some history. It would be interesting to maintain an overall elite path list that saves solutions from multiple generations. However evaluating each path at every iteration does get expensive.

```
double createRadialPath(PATH_STRUCT* pathway, int numNodes)
```

New function added for radial vision. Creates a new path and evaluates it based on a given number of nodes. In order to enable radial vision there must be a +Rad command line argument followed by the number of nodes to evaluate at a time. This function returns the fitness of this radial path.

7.3.1 Command Line Arguments

```
User Manual
A number of command line arguments can be added within a chosen execution file
or to the terminal in order to control the system. The following lists arguments
known to be in use.  If an argument is left out, a default value is set for the
corresponding paramater.


+Efi<FileName.out>: controls which benchmark (map) file is read from the bench
                  mark directory (example +Efiz1.out).  Benchmark file directory
                  is PP_ga_dr/TEST_dr/

+Efd<FileName.txt>: no longer used.  Old method of specifying benchmark file

+Efo<FileNamePrefix>: Controls the output filename prefix of the file read by
                  DisplayPaths.m in MatLab

+Ern<integer>:    Number of runs
```

+Ero<*1 or 0*>:	Method of updating ElitePaths[]. 1 = original, 0 = modified
+Ppa<*1 or 6*>:	Algorithm to use. 1 = GA, 6 = PSO
+Gpn<*integer>:*	Population size
+Gmg<*integer>:*	Maximum number of iterations to run
+Gif<*integer*>:	Injection Frequency (for GA). (Possibly no longer used)
+Gcv<*float*>:	Safe clearance distance
+GRR<*float>:*	Repair Ration (for GA)
+Gmt<*float>:*	Maximum Desired Turning Angle
+Pwd<*float>:*	Controls the effect of distance on the fitness value of a path
+Pws<*float>:*	Controls the effect of smoothness on the fitness value of a path
+Pwc<*float>:*	Controls the effect of clearance on the fitness value of a path
+Grx<*float 0 - 1>:*	Controls the crossover rate in the GA
+Grm<*float>:*	Controls the mutation rate in the GA
+Gri<*float>:*	Rate of improvement (shortcut) operator (GA)
+Grs<*float>:*	Rate of smoothen operator (GA)
+Guc<*integer*>:	Max number of iterations without improvement (GA). May no longer be used
+Gtg<*integer*>:	“Trace Gen”. Unknown function
+Gir<*integer>:*	“Inject Random”. Unknown function
+Gim<*integer*>:	“Inject Memory”. Unknown function
+Pds<*float>:*	Dynamic Severity. Affects distance by which dynamic obstacles (and robot?) move at each iteration.
+Pdf<*integer* –1 to 2>:	Dynamic Flag. Exact function unknown
+Pdr<*float*>:	Dynamic Rate. Affects number of iterations between dynamic obstacle movement
+Pdv<*float>:*	Robot velocity. Affects distance moved by robot at each robot move
+Pdg<*integer*>:	Number of iteration between dynamic environment update
+Pwa<*float>:*	Controls the effect of old collision avoidance cost
+Din<integer>*:*	Affects number of paths displayed in MatLab when using DisplayPaths()
+Rad<*integer>:*	Radial vision number of nodes in radial path. 0 turns off radial vision

+Spn<*integer*>:	Number of nodes in paths (PSO)
+Sps<*float*>:	Shortcut Rate (PSO)
+Spr<*float*>:	Disperse Nodes Rate (PSO)
+Svx<*float*>:	Maximum X Velocity (PSO)
+Svy<*float*>:	Maximum Y Velocity (PSO)
+Svi<*float*>:	Inertia (PSO)
+Sal<*float*>:	Local Best Attraction (PSO)
+Sag<*float*>:	Global Best Attraction (PSO)