MULTIPLE ROBOT CO-ORDINATION USING PARTICLE SWARM OPTIMISATION AND BACTERIA FORAGING ALGORITHM

A Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of

B. Tech.
(Mechanical Engineering)

By

PRASANNA K.
Roll. No. 10603056
and
SAIKISHAN DAS
Roll No. 10603062

Under the supervision of

DR. DAYAL R. PARHI
Professor
Department of Mechanical Engineering, NIT, Rourkela

DEPARTMENT OF MECHANICAL ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY
ROURKELA

MAY,2010

MULTIPLE ROBOT CO-ORDINATION USING PARTICLE SWARM OPTIMISATION AND BACTERIA FORAGING ALGORITHM

A Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of

B. Tech.

(Mechanical Engineering)

By

PRASANNA K.

Roll. No. 10603056

and

SAIKISHAN DAS

Roll No. 10603062

Under the supervision of

Dr. Dayal R. Parhi

Professor

Department of Mechanical Engineering, NIT, Rourkela

Department of Mechanical Engineering

National Institute of Technology

Rourkela

MAY, 2010

National Institute of Technology
Rourkela

C E R T I F I C A T E

This is to certify that the work in this thesis entitled **MULTIPLE ROBOT CO-ORDINATION USING PARTICLE SWARM OPTIMISATION AND BACTERIA FORAGING ALGORITHM** by **Saikishan Das (Roll No. 10603062)** has been carried out under my supervision in partial fulfillment of the requirements for the degree of **Bachelor of Technology** in ***Mechanical Engineering*** during session 2009- 2010 in the Department of Mechanical Engineering, National Institute of Technology, Rourkela.

To the best of my knowledge, this work has not been submitted to any other University/Institute for the award of any degree or diploma.

Dr. Dayal R. Parhi
(Supervisor)
Professor
Dept. of Mechanical Engineering
National Institute of Technology
Rourkela - 769008

ACKNOWLEDGEMENT

I would like to express my deep sense of gratitude and respect to my supervisor Prof. **Dayal R. Parhi** for his excellent guidance, suggestions and support. I consider myself extremely lucky to be able to work under the guidance of such a dynamic personality.

Last, but not the least I extend my sincere thanks to other faculty members of the Department of Mechanical Engineering, NIT Rourkela, for their valuable advice in every stage for successful completion of this project report.

DATE:
PLACE: N.I.T. ROURKELA

SAIKISHAN DAS
Roll No. - 10603062
Mechanical Engineering Department

CONTENTS

A B S T R A C T

The use of multiple robots to accomplish a task is certainly preferable over the use of specialised individual robots. A major problem with individual specialized robots is the idle-time, which can be reduced by the use of multiple general robots, therefore making the process economical. In case of infrequent tasks, unlike the ones like assembly line, the use of dedicated robots is not cost-effective. In such cases, multiple robots become essential. This work involves path-planning and co-ordination between multiple mobile agents in a static-obstacle environment. Multiple small robots (swarms) can work together to accomplish the designated tasks that are difficult or impossible for a single robot to accomplish. Here Particle Swarm Optimization (PSO) and Bacteria Foraging Algorithm (BFA) have been used for coordination and path-planning of the robots. PSO is used for global path planning of all the robotic agents in the workspace. The calculated paths of the robots are further optimized using a localised BFA optimization technique. The problem considered in this project is coordination of multiple mobile agents in a predefined environment using multiple small mobile robots. This work demonstrates the use of a combinatorial PSO algorithm with a novel local search enhanced by the use of BFA to help in efficient path planning limiting the chances of PSO getting trapped in the local optima. The approach has been simulated on a graphical interface.

LIST OF FIGURES AND TABLES

List of Figures.

.

List of Tables

.

.

.

CHAPTER 1

Chapter 1

INTRODUCTION

Use of a team of robots can help in monitoring, surveillance, and search and rescue operations, thus removing the need for human intervention in dangerous areas[1]. A simple example is exploration and search of an earthquake-hit building where each robot has a sensor(s) that can detect heat, light, sound, or other, and communicate wirelessly with other robots. Material handling and bomb detections are several other such aspects where multiple robots can co-ordinate among themselves to achieve required goal. Because of several desirable and undesirable constraints, resources must be distributed across multiple robots which must work in unison to accomplish a mission. Specialization of robot functions and collaboration amongst the deployed robots is employed to deal with these constraints.

In a static environment, use of multiple robots to accomplish a task with several complexities involves two important aspects- path planning and efficient robot co-ordination [2]

1.1. OBJECTIVE

The main objective of the work is to accomplish co-ordination between multiple robots in a known environment with static obstacles. This has to be achieved by doing path-planning for the robots using two very common swarm intelligence optimization techniques – PSO (for global search) and BFA (for local search).

1.2. PATH PLANNING

Path planning [3] is one of the fundamental problems in mobile robotics. According to Latombe [4], the capability of effectively planning its motions is "eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world."

Trajectory for each robot has to be computed in order to avoid collisions between the robots. Several undesirable situations like congestions and deadlocks may obstruct the progress of the robots. In such cases use of particle swarm optimization techniques can be used for efficient path planning and avoiding such undesirable situations. Particle Swarm Optimization (PSO) and Bacteria Foraging Algorithm (BFA) can be used in path planning and robot co-ordination.

1.3. PARTICLE SWARM OPTIMIZATION

PSO techniques are algorithms used to find a solution to an optimization problem in some search space [5-6]. PSO has been used for hazardous target search applications, such as landmine detection, fire fighting, and military surveillance, and is an effective technique for collective robotic search problems. When PSO is used for exploration, this algorithm enables robots to travel on trajectories that lead to total swarm convergence on some target. Two basic approaches to controlling multiple robots to achieve collective tasks are centralized control and distributed control. The PSO algorithm [5] can work in both centralized control and distributed control scenarios. In centralized control the robots are organized in a hierarchical fashion similar to the military; e.g. teams of robots are controlled by designated robot leaders which are controlled by the head robot for the entire swarm. The robots send pose information to the head robot which executes the PSO algorithm and returns new directional information to each robot. In decentralized control, each robot operates on local information but works toward accomplishing a global goal.

A decentralized PSO algorithm is used in this project for robots to find targets at known locations in an area of interest. Some issues in design and implementation of an unsupervised distributed PSO algorithm for target location include robot dispersion and deployment, localization, obstacle avoidance, overshooting targets, effect of particle neighbourhood sizes on performance and scalability.

1.4. BACTERIA FORAGING ALGORITHM

BFA is based on the foraging behaviour of *Escherichia Coli* (E. Coli) bacteria present in the human intestine and already been in use to many engineering problems. Studies by A. Stevens show that BFA is better than PSO algorithm in terms of convergence, robustness and precision. BFA is the latest trend that is efficient in optimizing parameters of the structures.

In this work, BFA has been used in the program developed using PSO in order to overcome the drawbacks and to help in efficient path planning limiting the chances of PSO getting trapped in the local optima.

Creation of complex virtual worlds and simulation of the robots in such environments can be done using C++ compiler [7]. A complete programming library is provided to allow users to program the robots C++ compiler. From the controller programs, it is possible to read input values and show the required simulation in a graphic window.

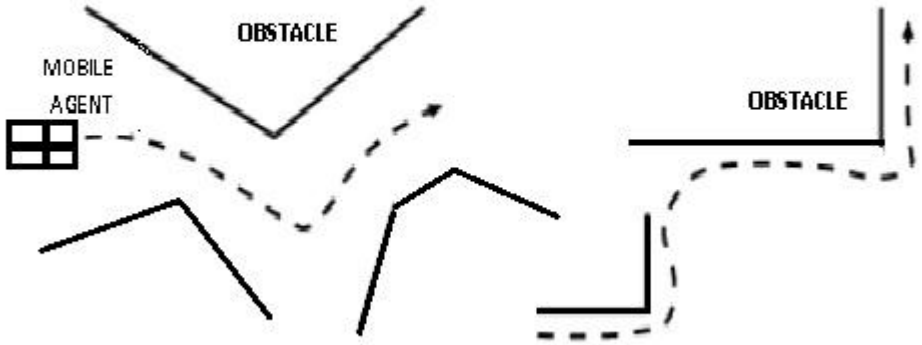

Fig.1. Path planning by using BFA

1.5. C++ COMPILER AND GRAPHICS

The compiler provides an environment for programing. The simulation of the co-ordination can be done using Windows MFC and graphics can be incorporated in order to graphically visualise the animation and simulation. It is crucial for the display of the output on the window screen.

*****

CHAPTER 2

Chapter 2

LITERATURE REVIEW

This section provides an insight and literature review to the current methodologies applied for co-ordination of multiple robot systems. It also highlights various methods used by researchers and their outcomes related to such problems.

2.1. ROBOT CO-ORDINATION

Several works have been done in the past and is going on in the field of multiple-robot co-ordination. Yamauchi [8] developed a distributed, asynchronous multi-robot exploration algorithm which introduces the concept of frontier cells. Frontier cells are the border areas between known and unknown environments. Their basic idea is to let each robot move to the closest frontier cell independently. This brings the fault tolerance capability. However, the multiple robots may move towards the same frontier cell, thus rendering the process ineffective. Therefore their algorithm lacks sufficient co-ordination.

Parhi *et al.* [9] proposed a control technique for navigation of intelligent mobile robots. Cooperative behaviours using a colony of robots are becoming more and more significant in industrial, commercial and scientific application. Problems such as co-ordination of multiple robots, motion planning and co-ordination of multiple robotic systems are generally approached having a central (hierarchical) controller in mind. Here by using Rule base technique and petri net modelling to avoid collision among robots one model of collision free path planning has been proposed. The second model incorporates rule based fuzzy-logic technique and both the models are compared. It has been found that the rule-based technique has a set of rules obtained through rule induction and subsequently with manually derived heuristics. This technique employs rules and takes into account the distances of the obstacles around the robots and the bearing of the targets in order to compute the change required in steering angle. With

the use of Petri net model the robots are capable of negotiate with each other. It has been seen that, by using rule-based-neuro-fuzzy technique the robots are able to avoid any obstacles (static and moving obstacles), escape from dead ends, and find targets in a highly cluttered environments. Using these techniques as many as 1000 mobile robots can navigate successfully neither colliding with each other nor colliding with obstacles present in the environment. It was observed that the rule-based-neuro-fuzzy technique is the best compared to the rule-based technique for navigation of multiple mobile robots.

Grabowski et al. [10] investigated the coordination of a team of miniature robots that exchange mapping and sensor information. In their system, a robot plays as a team leader that integrates the information gathered by the other robots. This team leader directs the movement of other robots to unknown areas. They developed a novel localization system that uses sonar-based distance measurements to determine the positions of all the robots in the group. With their positions known, an occupancy grid Bayesian mapping algorithm can be used to combine the sensor data from multiple robots with different sensing modalities.

Simmons et al. [11] developed a semi-distributed multi-robot exploration algorithm which requires a central agent to evaluate the bidding from all the other robots to obtain the most information gain while reducing the cost, or the travelling distance. However, there are a few limitations to this approach. The work has been done assuming that the robots begin in view of one another and are told their initial (approximate) relative location. But once the robots need to merge maps with initial co-ordinates unknown and with the aim to find out where they are relative to one another, more sophisticated techniques are needed for mapping and localization.

Gerkey and Mataric proposed an auction method for multi-robot coordination in their MURDOCH system [12]. A variant of the Contract Net Protocol, MURDOCH produces a distributed approximation to a global optimum of resource usage. The use

of an "auctioneer" agent is similar to the central agent method used in Simmons et al.'s work. The work basically shows the effectiveness of distributed negotiation mechanisms such as MURDOCH for coordinating physical multi-robot systems. In most of the previous work, the communication between robots is assumed to be perfect, which makes their algorithms unable to handle unexpected, occasional communication link breakdowns.

2.2. PATH PLANNING

Path planning for multiple robots [13] has been studied extensively over the past ten years. The path planning problem in robotics is to generate a continuous path for a given robot between an initial and a goal configuration (or placement) of the robot. Along this path, the robot must not intersect given forbidden regions (usually, obstacles) [4,14]. There are two basic approaches to the multi-robot path planning problem - centralised and decoupled. In case of centralised approach, each robot is treated as one composite system, and the planning is done in a composite configuration space, formed by combining the configuration spaces of the individual robots. Whereas, in case of decoupled approach, paths are first generated for the separate robots independently, and then their interactions are considered (with respect to the generated paths).

The advantage in case of centralised approaches is that they always find a solution when one exists. However, the practical difficulty is that, if completeness is to be obtained, it yields methods whose time complexity is exponential in the dimension of the composite configuration space. But decoupled planners inherently are incomplete and can lead to deadlock situations. Even the apparently simple problem of motion planning for arbitrarily many rectangular robots in an empty rectangular workspace is still PSPACE-complete [14].

2.2.1. Centralised planning: Ardema and Skowronski [15] described a method for generating collision-free motion control for two specific, constrained manipulators, by modelling the problem as a non-cooperative game.

Bennewitz *et al.* [3] presented a method for finding and optimizing priority schemes for such prioritized and decoupled planning techniques. The existing approaches apply a single priority scheme making them overly prone to failure in cases where valid solutions exist. By searching in the space of prioritization schemes, their approach overcomes this limitation. It performs a randomized search with hill-climbing to find solutions and to minimize the overall path length. To focus the search, algorithm is guided by constraints generated from the task specification. The experiments conducted not only resulted in significant reduction of the number of failures in which no solution can be found, they also showed a significant reduction of the overall path length.

Other centralised approaches for tackling multi-robot planning problems include various Potential Field Techniques. These techniques apply different approaches to deal with the problem of local minima in the potential function. Other methods restrict the motions of the robots to reduce the size of the search space. Tournassoud [16] proposed a potential field approach where the motion coordination is expressed as a local optimisation problem.

In [17] Barraquand *et al.* present a potential field technique for many discs in environments with narrow corridors. To escape local minima, so-called constrained motions are executed which force one configuration coordinate to increase or decrease until a saddle point of the potential function is attained. This potential field planner has been successfully experimented with for up to ten robots.

2.2.2. Decoupled planning: Decoupled planners help in finding the paths of the individual robots independently before employing different strategies to resolve possible conflicts. They may fail to find a solution even if there is one. A popular decoupled approach is planning in the configuration time-space by Erdmann and

Lozano-Perez [18], which can be constructed for each robot given the positions and orientations of all other robots at every point in time. Techniques of this type assign priorities to the individual robots and compute the paths of the robots based on the order implied by these priorities.

Ferrari *et al.* [19] uses a fixed priority scheme and chooses random detours for the robots with lower priority. Variations of initial solutions for collision-free robot paths are obtained with respect to quality parameters that give heuristic rules for evaluating plan robustness. While collision impact factors (CIF and CAF) are considered for evaluating the quality of a single path, performance indices (RT, ME and VE), are used for evaluating the overall quality of a plan.

Erdmann and Lozano-Perez [18] proposed the scheme of prioritised planning. The foremost task is to assign priorities to robots which is followed by picking up of the robots in order of decreasing priority. For each picked robot a path is planned, avoiding collisions with the static obstacles as well as the previously picked robots, which are considered as moving obstacles.

Another approach to decoupled planning is the path coordination method. The key idea of this technique is to keep the robots on their individual paths and let the robots stop, move forward, or even move backward on their trajectories in order to avoid collisions. Bien and Lee [16] proposed a method to achieve a path-constrained minimum time trajectory pair for both the robots with limited actuator torques and velocities. A two-dimensional coordination space is constructed to identify a collision region along the paths which is transformed into one in time-versus-travelled-length space with the velocity profile of one of the two robots.

2.3. PARTICLE SWARM OPTIMISATION

PSO is relatively a new concept reported by Kennedy and Eberhart (2001) [31], in 1995 and is often applied for tracing the targets by autonomous communicating bodies (Gesu *et al.,* 2000). PSO is a population based stochastic optimization

technique inspired by social behaviour of bird flocking or fish schooling. A problem space is initialized with a population of random solutions in which it searches for the optimum over a number of generations/iterations and reproduction is based on prior generations. The concept of PSO is that each particle randomly searches through the problem space by updating itself with its own memory and the social information gathered from other particles. In this work, the PSO particles are referred to as robots and the local version of the PSO algorithm is considered in the context of this application (Kennedy, 1999). An extensive search on Particle Swarm Optimisation has been carried out by Polli (2007) [21]. PSO has been used by researchers all over the world from various fields of research for different types of optimisation.

In the work done by Ray *et al*.(2010) [22], selected lower order harmonics of multilevel inverter are eliminated while the overall voltage THD (Total Harmonic Distortion) is optimized by computing the switching angles using Particle Swarm Optimization (PSO) technique.

In [23], N.M. Kwok *et al.* (2009) proposed an improved PSO model for solving the optimal formation reconfiguration control problem for multiple UCAVs. The proposed strategy can produce a large speed value dynamically according to the variation of the speed, which makes the algorithm explore the local and global minima thoroughly at the same time. Series experimental results demonstrate the feasibility and effectiveness of the proposed method in solving the optimal formation reconfiguration control problem for multiple UCAVs.

Most recently, A. Atyabi *et al.* (2010) [24] employed two enhanced versions of PSO - area extension PSO (AEPSO) and cooperative AEPSO (CAEPSO) as decision makers and movement controllers of simulated . The study examines the feasibility of AEPSO and CAEPSO on uncertain and time-dependent simulated environments.

2.4. BACTERIA FORAGING ALGORITHM

In 2002, K. M. Passino proposed Bacterial Foraging Optimization Algorithm (BFOA) for distributed optimization and control. BFA is based on the foraging behaviour of Escherichia Coli (*E. Coli*) bacteria present in the human intestine [25] and already been in use to many engineering problems including multiple robot co-ordination. According to paper [26], BFA is better than Particle Swarm Optimisation in terms of convergence, robustness and precision.

T.Datta *et al.(*2008) [26] proposed an improved adaptive approach involving Bacterial Foraging Algorithm (BFA) to optimize both the amplitude and phase of the weights of a linear array of antennas for maximum array factor at any desired direction and nulls in specific directions.

Tang W.J. *et al.* (2008) [27] presented a new algorithm, dynamic bacterial foraging algorithm (DBFA),based originally on the BFA for solving an OPF(Optimal Power Flow) problem in a dynamic environment in which system loads are changing. He concluded that DBFA can more rapidly adapt to load changes, and more closely traces the global optimum of the system fuel cost, in comparison with BFA and particle swarm optimizer.

A.Dhariwal *et al.* (2004) [28] presented an approach, inspired by bacterial chemotaxis, for robots to navigate to sources using gradient measurements and a simple actuation strategy (biasing a random walk). They have showed the efficacy of the approach in varied conditions including multiple sources, dissipative sources, and noisy sensors and actuators through extensive simulations.

2.5. HYBDRID AND COMBINATORIAL APPROACH

Several works have been done in the field of path planning for multiple robots using the PSO and BFA algorithms. But recently, researchers are focussing on hybrid or combinatorial optimisation techniques [23][29][30], which incorporate two or more

optimisation techniques together in order to avoid several undesirable problems faced by previous researchers.

For example PSO is used for global path planning of all the robotic agents in the workspace. The calculated paths of the robots are further optimized using a localised BFA optimization technique. This helps in better convergence of results.

Recently, superior results have been obtained in proportional integral derivative controller tuning application by using a new algorithm BFOA oriented by PSO termed BF-PSO. This study conducted by Hai Shen *et al.* (2009) [29] shows that

BFPSO performs much better than BFOA for almost all test functions.

A.Biswas *et al.* [30] has presented an improved variant of the BFOA algorithm by combining the PSO based mutation operator with bacterial chemotaxis. The work judiciously uses the exploration and exploitation abilities of the search space, hence, avoiding undesirable and false convergence. The proposed algorithm has far better performance than a standalone BFOA at least on the numerical benchmarks tested.

Due to robot localization, the system is partially dynamic and information for fitness evaluation is incomplete and corrupted by noise. Kwok *et al*. (2006) [23] applied three evolutionary computing techniques, including genetic algorithms (GA), particle swarm optimization (PSO) and ants system (AS) to the localization problem. Their performances are compared based on simulation and experiment results and the feasibility of the proposed approach to mobile robot localization is demonstrated.

Our objective is the co-ordination of several robots in a predetermined static environment. We have applied both PSO for global search and BFA for local optima. BFA has been used to help in efficient path planning limiting the chances of PSO getting trapped in the local optima.

******

CHAPTER 3

Chapter 3

ANALYSIS

3.1.PROBLEM FORMULATION

The sample environment consists of a rectangular space comprising stationary obstacles about which we have priori knowledge. The environment information includes the limits of the rectangular workspace and the shape, location and orientation of all the stationary obstacles in the given workspace. The workspace is assumed to have no mobile obstacles.

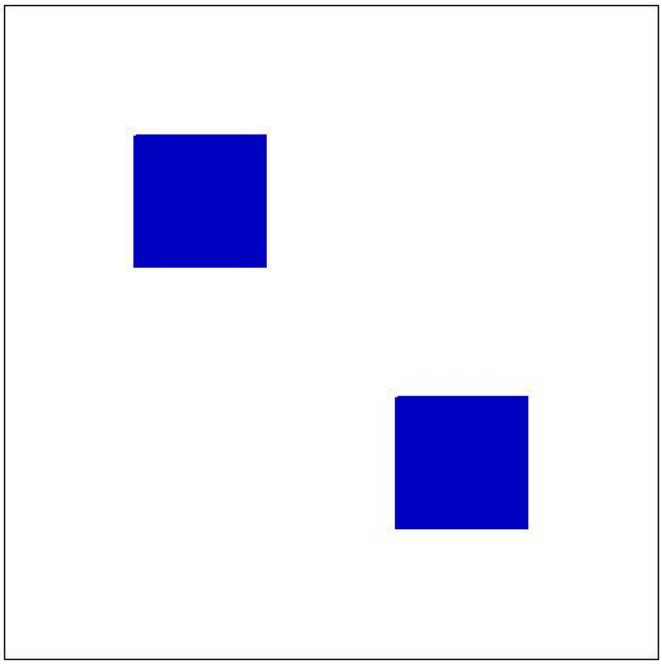

Fig.2. Sample Environment with stationary obstacles

Here the large rectangular unfilled boundary (Fig.2) represents the limits of the workspace. The solid polygons inside the workspace are the stationary obstacles. Each robot has a source and a goal point which is given at the start of the problem. Both the source and goal points lie within the limits of the workspace.

3.2. PARTICLE SWARM OPTIMISATION (PSO)

Particle swarm optimization (PSO) is a method for performing numerical optimization without explicit knowledge of the gradient of the problem to be optimized. Developed by Kennedy and Eberhart in 1995 [31], PSO is a population based stochastic optimization technique inspired by the social behaviour of bird flock and fish schools As a relatively new evolutionary paradigm, it has grown in the past decade and many studies related to PSO have been published.

The algorithmic flow in PSO starts with a population of particles whose positions, that represent the potential solutions for the studied problem, and velocities are randomly initialized in the search space. The search for optimal position (solution) is performed by updating the particle velocities, hence positions, in each iteration/generation in a specific manner follows.

Reynolds in 1987 [32] proposed a behavioural model in which each agent follows three rules below.

- Separation- Each agent tries to move away from its neighbours if they are too close.
- Alignment- Each agent steers towards the average heading of its neighbours.
- Cohesion- Each agent tries to go towards the average position of its neighbours.

Fig.3.Graphical representation of separation, alignment and cohesion.

Kennedy and Eberhart [37] included a ‘roost’ in a simplified Reynolds-like simulation

so that:

- Each agent was attracted towards the location of the roost.
- Each agent 'remembered' where it was closer to the roost.
- Each agent shared information with its neighbours (originally, all other agents) about its closest location to the roost.

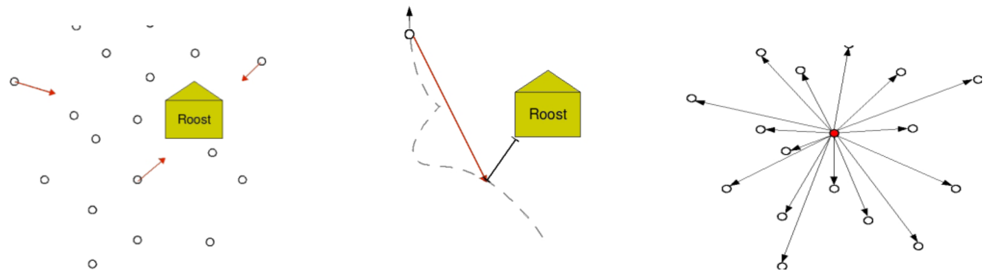

Fig.4. Graphical representation of Kennedy and Eberhart model.

Eventually, all agents 'landed' on the roost. If the notion of distance to the roost is changed by an unknown function, the agents will 'land' in the minimum.

3.2.1. Basic Steps in PSO

Step 1. Create a 'population' of agents (called particles) uniformly distributed over X.

Step 2. Evaluate each particle's position according to the objective function.

Step 3. If a particle's current position is better than its previous best position, update it.

Step 4. Determine the best particle (according to the particle's previous best positions).

Step 5. Update particles' velocities according to

$$v_{ij}^{t} = wv_{ij}^{t-1} + c_1 r_1 (p_{ij}^{t-1} - x_{ij}^{t-1}) + c_2 r_2 (g_j^{t-1} - x_{ij}^{t-1}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (1)$$

Step 6. Move particles to their new positions according to

$$x_{ij}^{t} = x_{ij}^{t-1} + v_{ij}^{t} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (2)$$

Step 7. Go to step 2 until stopping criteria are satisfied.

The main variants in PSO are inertia (w), personal influence (c_1) and social influence (c_2) which refer to the corresponding terms in velocity update equation respectively.

Many improvements have been incorporated into this basic algorithm. One of the example of such a modification can be seen in [30].

3.2.2. Problem Implementation

Step 1: Link generation takes place i.e. all feasible links from each vertex point, source and goal to the other vertex points, source and goal are generated.
Step 2: The particles P_i are generated all of particle length zero starting from source point with no intermediate vertex points. Each particle has a set of intermediate vertex points given by C_j. The number of intermediate points keeps increasing as the particle propagates and grows. The length of particle P_i is specified in the variable L_i.

Particle P_i	$C_1,C_2,...,C_k$
Length L_i	K

Table 1. PSO particle structure

Step 3: The intermediate points are chosen from the set of linked points available to the source point. The linked point with the lowest objective value is chosen.

$$objective = \sqrt{(x-x_g)^2+(y-y_g)^2}$$

Where (x,y) it the coordinates of the linked point and (x_g,y_g) is the coordinates of the goal.
Step 4: The particle length of each particle increases as intermediate vertex points are included after the source point of the particle.
Step 5: New intermediate vertex points are added in the same way and the particle length increases with every addition.

Step 6: When a newly added intermediate point is linked to the goal, the goal point is added to the path.
Step 7: The process ends.

3.3. BACTERIA FORAGING ALGORITHM

The details of BFA are given in [31]. This algorithm is modeled on the foraging technique of a group of bacteria which move in search of food and away from noxious elements — this method is known as foraging. All bacteria try to ascend the food concentration gradient individually. The food concentration is measured at the initial location and then a tumble takes place assigning a random direction and swim for a given fixed distance and measure the concentration there. This tumble and swim make one chemotactic step. If the concentration is greater at next location then they take another step in that direction. When the concentration at next location is lesser that of previous location they tumble to find another direction and swim in this new direction. For a certain number of steps this process is carried out, which is limited by the lifetime of the bacteria. At the end of its lifetime the bacteria that have gathered good health that are in better concentration region divide into two cells. Thus in the next reproductive step the next generation of bacteria start from a healthy position. The better half reproduces to generate next generation where as the worse half dies. This optimization technique enables us to take the variable we want to optimize as the location of bacteria in the search plane (the plane where the bacteria can move).

The specifications such as number of reproductive steps, number chemotactic steps which are consisted of run (or swim) and tumble, swim length, maximum allowable swims in a particular direction are given for a particular problem then the variable can be optimized using this Bacteria Foraging Optimization technique.

Chemotaxis: Chemotaxis is achieved through swimming and tumbling. Depending on the food concentration, it decides whether it should move in a predefined direction (swimming) or an altogether different direction (tumbling), in the entire lifetime of the bacterium. A tumble is represented by a unit length random direction, φ(j) say, is generated; this will be used to define the direction of movement after a tumble. In particular,

$$\theta^i(j+1,k,l) = \theta^i(j,k,l) + C(i)\Phi(j)$$

where θ^i(j, k, l) represents the i^{th} bacterium at j^{th} chemotactic k^{th} reproductive, and i^{th} elimination and dispersal step. C(i) represents the size of the step taken in the random direction. "C" is termed as the "run length unit".

Swarming: It is desired for the bacterium that has searched the optimum path of food to attract other bacteria to itself so that they reach the desired place rapidly. Swarming results in congregation of bacteria into groups and hence move as concentric patterns of groups with high bacterial density. Mathematically, swarming can be represented as

$$J_{cc} = \sum_{j=1}^{S} J_{cc}^{i}(\theta, \theta^i(j,k,l))$$

$$= \sum_{j=1}^{S}[-d_{attract}\exp(-\omega_{attract}\sum_{m=1}^{p}(\theta_m - \theta_m^i)^2)] + \sum_{j=1}^{S}[h_{repellent}\exp(-\omega_{repellent}\sum_{m=1}^{p}(\theta_m - \theta_m^i)^2)]$$

Where J_{cc} *(θ, P(j, k, l))* is the cost function value to be added to the actual cost function to be minimized to present a time varying cost function. "S" is the total number of bacteria. "*p*" is the number of parameters to be optimized that are present in each bacterium. $d_{attract}$, $\omega_{attract}$, $h_{repellent}$, *and* $\omega_{repellent}$ are different coefficients that are to be chosen judiciously.

Reproduction: The healthiest bacteria reproduces into two bacteria and the least health ones die. Therefore the bacteria population remains constant.

Elimination and Dispersal: The life of a population of bacteria changes gradually by consumption of nutrients or abruptly due to other influences. Such instances can kill or disperse bacteria present in specific regions.

3.3.1. General Steps of BFA

Following shows the basic BF algorithm as proposed by Passino [31].

For initialization, we must choose p, S, N_c , N_s , N_{re} , N_{ed} , p_{ed} and the $C(i)$, $i = 1,2,K, S$. If we use swarming, we will also have to pick the parameters of the cell-to-cell attractant functions; here we will use the parameters given above. Also, initial values for the θ^i, $i = 1,2\ldots,S$, must be chosen. Choosing these to be in areas where an optimum value is likely to exist is a good choice. Alternatively, we may want to simply randomly distribute them across the domain of the optimization problem. The algorithm that models bacterial population chemotaxis, swarming, reproduction, elimination, and dispersal is given here (initially, $j = k = l = 0$). For
the algorithm, note that updates to the θ^i automatically result in updates to P. Clearly, we could have added a more sophisticated termination test than simply specifying a maximum number of iterations.

1) Elimination-dispersal loop Process: $l = l + 1$

2) Reproduction loop Process: $k = k + 1$

3) Chemotaxis loop Process: $j = j + 1$

a) For $i = 1,2\ldots,S$, take a chemotactic step for bacterium
i as follows.

b) Calculate $J(i,j,k,l)$. Let $J(i,j,k,l) = J(i,j,k,l) + J_{cc}(\theta^i(j\ k\ l), P(j\ k\ l))$ (i.e., sum up the cell-to-cell attractant effect to the nutrient concentration).

c) Let $J_{last} = J(i,j,k,l)$ to save this value as we might find a better cost during a run.

d) Tumble Process: Generate a random vector $\Delta(i) \in A^p$ with each element $\Delta_m(i), m = 1,2...,p$, a random number on $[-1,1]$.

e) Move: Let

$$\theta^i(j+1, k,l) = \theta^i(j,k,l) + C(i) * \{\Delta(i) / \{\Delta^T(i)\,\Delta(i)\}^{0.5}\}$$

This results in a step of size $C(i)$ in the direction of the tumble for bacterium i.

f) Calculate $J(i, j + 1,k,l)$, and then let $J(i, j + 1,k,l) = J_{cc}(\theta^i(j+1,k,l), P(j+1,k,l))$.

g) Swim (note that we use an approximation since we decide swimming behaviour of each cell as if the bacteria numbered $\{1,2\ldots i\}$ have moved and $\{i + 1, i + 2\ldots S\}$ have not; this is much simpler to simulate than simultaneous decisions about swimming and tumbling by all bacteria at the same time):

i) Let $m = 0$ (counter for swim length).

ii) While $m < N_s$ (if have not climbed down too long)

- Let $m = m + 1$.
- If $J(i, j + 1,k,l) < J_{last}$ (if doing better), let

  $J_{last} = J(i,j+1, k,l)$ and let

  $\theta^i(j+1,k,l) = \theta^i(j+1,k,l) + C(i) * \{\Delta(i) / \{\Delta^T(i)\,\Delta(i)\}^{0.5}\}$

  and use this $\theta^i(j+1,k,l)$ to compute the *new* $J(i, j + 1,k,l)$ as we did in (f).
- Else, let $m = N_s$

This is the end of the while statement.

h) Move to next bacterium $(i + 1)$ if $i \neq S$ (i.e., go to b) for processing the next bacterium).

4) If $j < N_c$, go to step 3. In this case, chemotaxis continues, as the life of the bacteria is not complete.

5) Reproduction phase:

a) For the given k and l, and for each $i = 1,2\ldots S$, let

$J^{i}_{\text{health}} = \sum_{j=1}^{N_c} J(i, j, k, l)$ be the overall health of the bacterium i (a measure of the content of nutrients it had got over its lifetime and how able it was at avoiding noxious substances). Arrange bacteria and chemotactic parameters $C(i)$ in ascending order of cost J_{health} (higher cost means lower health).

b) The S_r bacteria with the highest J_{health} values die and the other S_r bacteria with the best values reproduce (and the copies that are made are placed along side their parent).

6) If $k < N_{re}$, go to step 2. In this case, as we have not reached the specified number of reproduction steps, we start the next generation in the chemotactic loop.

7) Elimination-dispersal Phase: For $i = 1, 2 \ldots S$, with probability p_{ed} , will eliminate and disperse each bacterium (this keeps the the population of the bacteria constant).

On eliminating a bacterium, simply disperse a new one to a random location on the problem domain.

8) If $l < N_{ed}$, then go to step 1; otherwise end.

3.3.2. Problem Implementation

Step 1: The number of control points per PSO output path line segment v is pre-determined by the user and are represented as C_{ijk}.

Where C refers to the control point and the index i,j and k refer to the corresponding particle, segment and control point respectively.

Step 2: v control points are included into the pso output line segment such that each point is equally spaced and lies on the line segment.

Step 3: b number of bacteria B_j are generated are randomly generated each having the whole set of control points. The particle structure is as shown below:-

	Segment 1	Segment 2	.....	Segment n
Bacteria B_1	$C_{111},C_{112},...,C_{11v}$	$C_{121},C_{122},...,C_{12v}$	.....	$C_{1n1},C_{1n2},...,C_{1nv}$
Bacteria B_2	.....	.....	.....	.....
.....	.....	.....	.....	.....
Bacteria B_b	$C_{b11},C_{b12},...,C_{b1v}$	.....	.....	$C_{bn1},C_{bn2},...,C_{bnv}$

Table 2. BFA particle structure

Step 4: Each control point is allowed to move freely in eight possible directions (north,south,east,west,north-east,south-east,north-west and south-west) which correspond to the eight neighbourhood pixels of any point on screen respectively as shown below. This phase is called swimming phase.

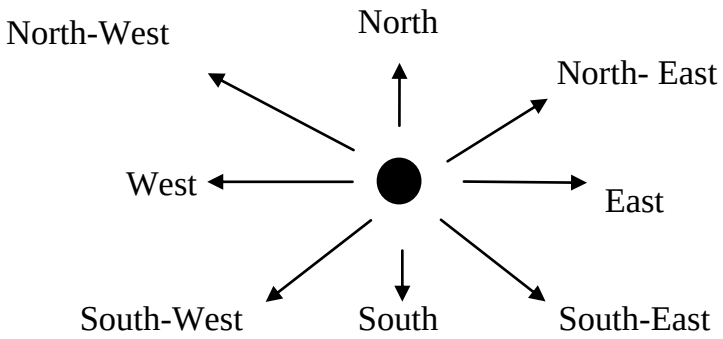

Fig.5. Possible Directions

Step 5: If the objective value of the particle decreases, the control points continue to move in the same direction. If the objective value increases, the last movement of the bacteria is retraced and a random direction from the available eight is chosen for the bacteria to move.

Step 6: This repeats till the life-time of the bacteria gets completed. Each bacteria movement constitutes one life-time unit. N life-time units comprise of a life-time of the bacteria.

Step 7: Next to the swimming phase is the swarming phase. Here the best bacterium among the lot is found out. All the other bacteria tend to move towards this best

bacterium. The corresponding control points between each bacterium and the best bacteria are compared and the control point of each bacterium is forced to swim towards the corresponding control point of the best bacterium. The number of swims in this phase is determined by the swarming life of each bacterium which is given by N'.

Step 8: The next phase is the reproduction phase where all the bacteria are first arranged in the ascending order of their objective values. The number of reproducing bacteria and number of offsprings per reproducer is determined by the relations:-

$$reproducers = reproducers\_ratio \times Population$$

$$offsprings/reproducer = \frac{offsprings\_ratio \times Population}{reproducers}$$

The best bacteria on the top of the stack of bacterium are chosen as reproducers. Each parent bacterium gives raise to the calculated number of offsprings. These offsprings replace the worst bacteria from the bottom of the stack with high objective values.

Step 9: Termination conditions are checked in this step. If any of the conditions are satisfied, the iteration terminates. If the program reaches the maximum number of termination iterations specified, the program terminates. Otherwise if the objective value of the global best bacterium does not vary for the specified number of repetition iterations, the iterations terminate.

******

CHAPTER 4

Chapter 4

RESULTS AND DISCUSSIONS

A hybrid PSO algorithm incorporating underlying procedures of potential field method has been successfully implemented to act as a path planning algorithm for coordinating multiple robots. The algorithm has been extensively tested on a variety of sample environments taken from literature. A sample environment is shown here in fig 6.

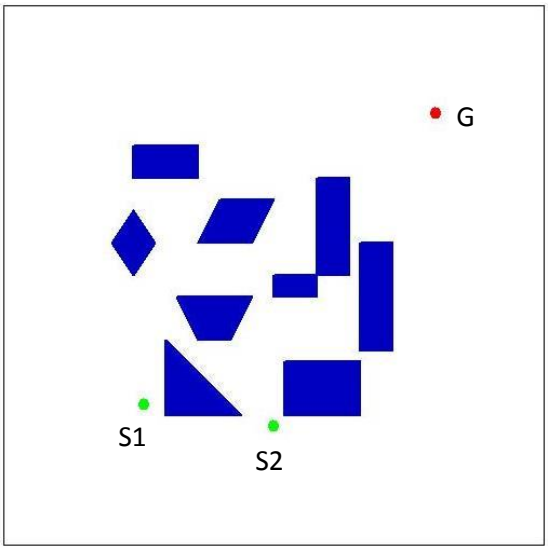

Fig.6. Sample Environment

Here the rectangular unfilled boundary signifies the limits of the workspace within which the robots can move. The blue solid polygons are stationary obstacles in the environment. The position, orientation and shape of the obstacle in the environment are input to the robot prior to its working in the workspace. The green points specify the starting points of robots. Since we are using two robots in this example, we have two starting points labelled S1 and S2. The red point labelled G is the specified destination for both the robots.

When modified PSO incorporating potential field is applied to solve this illustrated problem in fig 6. The results given by the algorithm are shown in fig 7. The solid black lines shown in fig 7 show the path given by the PSO algorithm. Here clearly the path calculated by PSO in fig 7 is seen to be a string of line segments between the source, goal and vertex points of the obstacles. When BFA is applied over the path calculated by PSO as input to it, it gives a further optimized path which can be seen in fig 8. Here the red line in fig 8 is the BFA's calculated path and the orange dots are the BFA control points of the path. The path consists of line segments connecting these control points.

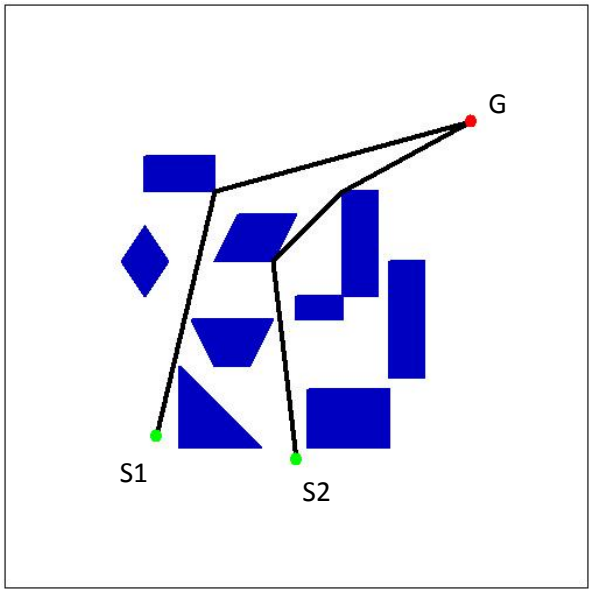

Fig.7. PSO calculated path

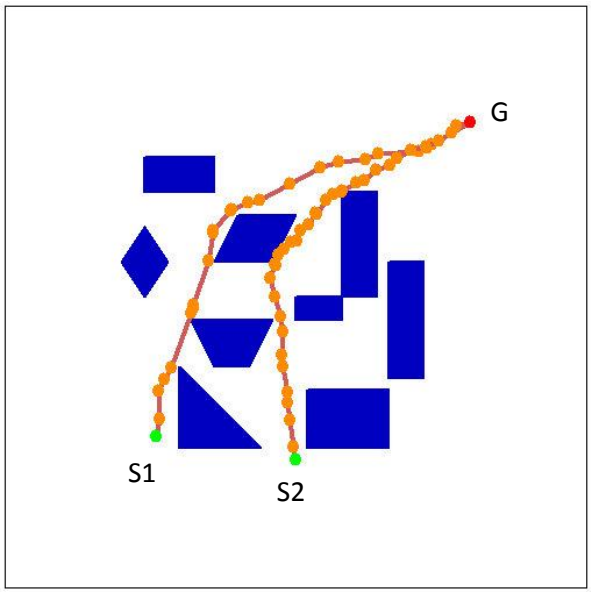

Fig.8. BFA calculated path

The different PSO+BFA parameters used are:-

Number of Particles:	10
Number of Bacteria:	10
Step Length:	1
Number of Directions:	8
Life Time of Bacteria:	5
Swarming Life of Bacteria:	2

Reproducers Ratio:	0.3
Offsprings Ratio:	0.4
Termination Iteration:	2000
Repetition Iteration:	500
Control Points/Segment:	12

It is clearly visible that the path obtained when BFA is applied over PSO is far better than the path obtained only by PSO. Fig 9 clearly illustrates this fact. The PSO path is shown by a black line and BFA by a red line. The BFA algorithm had delocalised itself from the previous intermediate vertex points of the PSO path and gives a close to smooth curve rather than straight lines. The increase in the number of intermediate points between the PSO and PSO+BFA algorithm in turn raises the degrees of freedom of the path contour which has been profitably exploited to give a path of greater accuracy. The numerical values of the length of the path calculated by PSO and PSO+BFA (in units) are shown in Table 3. The length of the path calculated again as shown in Table 1 clearly shows the superiority of PSO+BFA over PSO alone.

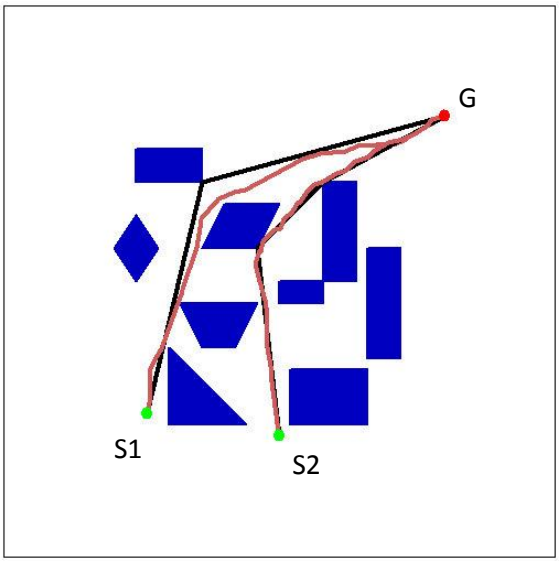

Fig.9. PSO and BFA path comparison

	Robot 1	Robot 2
PSO	443.9	296.5
PSO+BFA	430.7	387.5

Table 3. Optimum Path Length comparision between PSO and PSO+BFA

Visual Studio 2008 was used to develop the simulation. The programming language used was Visual C++. The simulation was run on a Pentium IV processor computer.

The algorithm can support any number of robots with each robot having its own source and goal. The optimum path calculated by this PSO+BFA algorithm can be profitable used for coordinating multiple robots in real-time industrial environments where there are pre-determined known obstacles in the workspace. A reduction in the length of the path travelled by robots directly translates into saving in cost and time. An army of general purpose mobile robots can efficiently function by coordinating among themselves to complete specialized tasks which would otherwise require dedicated specialized robots. Furthermore coordination among these general purpose robots will reduce the need of specialized robots leading to great savings in cost to industry and efficient utilization of the resources of available in workspace.

******

CHAPTER 5

Chapter 5

CONCLUSION

This project illustrates that the PSO algorithm based on potential field principles performs well in combinatorial optimization problems where a sequential combination of vertex points of obstacles constitute the path. This method of using a collection of vertex points as domain for the path restricts the solution space from the whole non-obstacle free space containing uncountable points to a discrete set of points. This drastic reduction in the problem domain points reduces the optimization time and computational resources required. But the reduction in the number of domain points compromises the optimum path calculated as the path generated by the PSO algorithm is in most cases not an optimum path but a path of lesser accuracy. To overcome this problem, BFA has been implemented over PSO and the results are found to have improved by a considerable extent. Now the intermediate points which were previously restricted to the vertices of the obstacles in the PSO algorithm are delocalised by the BFA control points, a specific number of which are included between each PSO path segment. The results of this PSO+BFA combine are shown to perform better than PSO alone.

PSO has a high convergence speed but is found to suffer in terms of accuracy. On the other hand BFA is a highly structured algorithm that has a poor convergence speed but high accuracy. A combination of PSO+BFA is hence endowed with high convergence speed and commendable accuracy. This can be otherwise stated as the PSO performing a global search and providing a near optimal path very quickly which is followed by a local search by BFA which fine-tunes the path and gives an optimum path of high accuracy. PSO has an inherent disability of trapping in the local optima but high convergence speed whereas BFA has the drawback of having a very poor convergence speed but the ability to not trap in the local optima. The PSO+BFA

combine gets the best of the both individual algorithms by having a good convergence speed and overcomes the disability of trapping in the local optima.

A mutation operator can be implemented with PSO to further enhance its accuracy and avoid it from trapping in the local optima. A similar dispersal operator when added to BFA will enhance its accuracy and efficiency. The same problem can be further extended to an environment with mobile obstacles. In that case online path planning has to be incorporated along with the offline algorithm with suitable sensors mainly ultrasonic, LIDAR, camera, proximity sensors, etc. to detect the position and location of the mobile obstacles. Localisation sensors will be required for both offline and online path-planning. Localisation and online path planning together will constitute Simultaneous Localisation and Mapping Algorithms (SLAM). Extensive research work in SLAM and related algorithms can be developed while implementing this code in robots in real-time. The robots will be capable of working in environments with priori-knowledge like shop-floor and unknown environments like open terrain.

******

REFERENCES

[1] Kurt Derr and Milos Manic, *Multi-Robot, Multi-Target Particle Swarm Optimization Search in Noisy Wireless Environments"*, Catania, Italy, May 21-23, 2009.

[2] W. Sheng, QingyanYang, Jindong Tan, Ning Xid, *"Distributed multi-robot coordination in area exploration*", Robotics and Autonomous Systems 54 (2006) 945–955.

[3] Maren Bennewitz, Wolfram Burgard, Sebastian Thrun, *"Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots"*, Robotics and Autonomous Systems, Volume 41, Issues 2-3, 30 November 2002,Pages89-99.

[4] J. Latombe, "*Robot Motion Planning"*, Kluwer Academic.

[5] J.Kennedy and Eberhart R.C., *"Particle swarm optimization",* Proc. IEEE International Conference on Neural Networks, IV, Piscataway, NJ, pp. 1942-1948,1995.

[6] Craig W. Reynolds, *"Flocks, herds, and schools: A distributed behavioral model*", ACM Computer Graphics, 21(4):25–34, 1987.

[7] Sanmay Satpathy*, "Controling of mobile agents using intelligent strategy",* NIT Rourkela 2009.

[8] Brian Yamauchi, *"Frontier-Based Exploration Using Multiple Robots*", Navy Center for Applied Research in Artificial Intelligence.

[9] Dayal Ramakrushna Parhi, Saroj Kumar Pradhan, Anup Kumar Panda, Rabindra Kumar Behera, *"The stable and precise motion control for multiple mobile robots"*, Applied Soft Computing 9 (2009) 477–487.

[10] R. Grabowski, L. Navarro-Serment, C. Paredis and P. Khosla, *"Heterogeneous teams of modular robots for mapping and exploration*", Journal of Autonomous *Robots* 8 (2000) (3), pp. 293–308.

[11] R. Simmons, D. Apfelbaum, W. Burgard, D. Fox, S. Thrun, H. Younes, *"Coordination for multi-robot exploration and mapping"*, Proceedings of the National Conference on Artificial Intelligence, 2000.

[12] B.P. Gerkey, M.J. Mataric, *"Sold Auction methods for multirobot coordination",* IEEE Transactions on Robotics and Automation 18 (2002) (5), pp. 758–768

[13] P. Svestka, M.H. Overmars, *"Coordinated path planning for multiple robots*", Robotics and Autonomous Systems 23 (1998) 125-152.

[14] J. Hopcroft, J.T. Schwartz, M. Sharir, *"Complexity of motion planning for multiple independent objects-PSPACE-hardness of the warehouseman's problem*", International Journal of Robotics Research 3 (4) (1984) 76--88.

[15] M.D. Ardema, J.M. Skowronski, *"Dynamic game applied to coordination control of two arm robotic system",* R.P. Hamalainen. H.K. Ehtamo (Eds.), Differential Games - Developments in Modelling and Computation, Springer, Berlin, 1991, pp. 118-130.

[16] P. Tournassoud*, "A strategy for obstacle avoidance and its application to multi-robot systems"*, Proceedings of the IEEE International Conference on Robotics and Automation, San Francisco, CA, USA, 1986, pp. 1224-1229.

[17] J. Barraquand, B. Langois and J.C. Latombe, *"Numerical potential field techniques for robot path planning",* IEEE Transactions on Robotics and Automation, Man and Cybernetics 22 2 (1992), pp. 224.

[18] M. Erdmann and T. Lozano-Perez, *"On multiple moving objects*", Algorithmica 2 (1987), pp. 477–521.

[19] C.Ferrari, E.Pagello, J. Ota and T.Arai , *"Multirobot motion coordination in space and time",* Robotics and Autonomous Systems 25 (1998), pp. 219–229.

[20] Z. Bien and J. Lee, *"A minimum-time trajectory planning method for two robots"* IEEE Transactions on Robotics and Automation 8 3 (1992), pp. 414–418.

[21] Poli, R., *"An analysis of publications on particle swarm optimisation applications"*, Technical Report CSM-469 (Department of Computer Science, University of Essex, UK). (2007).

[22] Ray, R.N., Chatterjee, D., Gowami, S.K., *"A PSO based Optimal Switching Technique for Voltage Harmonic Reduction of Multilevel Inverter",* Expert Systems with Applications (2010)

[23] N.M. Kwok et al., *"Evolutionary computing based mobile robot localization",* Engineering Applications of Artificial Intelligence 19 (2006) 857–868

[24] A. Atyabi et al., *"Navigating a robotic swarm in an uncharted 2D landscape",* Applied Soft Computing 10 (2010) 149–169

[25] Passino, K. M*., "Biomimicry of bacterial foraging for distributed optimization and control," ,* IEEE Control Systems Magazine, Vol. 22, No. 3, 52–67, June 2002.

[26] T. Datta *et al. "Improved Adaptive Bacteria Foraging Algorithm In Optimization Of Antenna Array For Faster Convergence",* Progress In Electromagnetics Research C, Vol. 1, 143–157, 2008

[27] Tang W.J. *et al. "Bacterial Foraging Algorithm for Optimal Power Flow in Dynamic Environments",* IEEE Transactions On Circuits And Systems—I: Regular Papers, Vol. 55, No. 8, September 2008.

[28] Dhariwal, Amit *et al., "Bacterium-inspired Robots for Environmental Monitoring"*, lnterrnational Conference on Robotics and Automation New Orleans. LA, 2004

[29] Hai Shen, Yunlong Zhu, Xiaoming Zhou, Haifeng Guo, Chunguang Chang; *"Bacterial Foraging Optimization Algorithm with Particle Swarm Optimization Strategy for Global Numerical Optimization";* 2009, Shanghai, China.

[30] Arijit Biswas, Sambarta Dasgupta, Swagatam Das, Ajith Abraham, "*Synergy of PSO and Bacterial Foraging Optimization –A Comparative Study on Numerical Benchmarks"*, Innovations in Hybrid Intelligent Systems, ASC 44, pp. 255–263, 2007.

[31] J.Kennedy and Eberhart R.C., *"Particle swarm optimization",* Proc. IEEE International Conference on Neural Networks, IV, Piscataway, NJ, pp. 1942-1948,1995.

[32]Craig W. Reynolds, *"Flocks, herds, and schools: A distributed behavioral model*", ACM Computer Graphics, 21(4):25–34, 1987.

[33] Jeff Prosise, "*Programming Windows with MFC",* Microsoft Press.

****

APPENDIX

SOURCE CODE

A hybrid robot navigation application in Visual C++ has been which integrates PSO and BFA for optimization and path planning. Microsoft Foundation Class (MFC) is used for creating the graphics features of the application.

a. Data Structure(STRUCTURES.h)
b. MFC Classes(HELLO.h)
c. MFC Application Initialiser(HELLO.cpp)
d. Map class(MAP.h)
e. Map information(MAP.cpp)
f. PSO class(PSO.h)
g. PSO path optimization(PSO.cpp)
h. BFA Class(BFA.h)
i. BFA Path Optimisation(BFA.cpp)
j. Geometry class(GEOMETRY.h)
k. Geometry of obstacles(GEOMETRY.cpp)
l. Graphics Classes(DRAW.h)
m. Graphics Functions(DRAW.cpp)
n. Map Data(MAP.txt)
o. Source Goal Data(SOUCE GOAL.txt)

****

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: STRUCTURS.h
Intent: Data structures
*****************************************************************/

#ifndef _STRUCTURES_H
#define STRUCTURES_H
#pragma once

#define ZERO 0.01
#define TOTAL_OBSTACLES 100
#define TOTAL_VERTEX 6
#define TOTAL_CONTROL_PTS 12
#define TOTAL_PARTICLES 1
#define TOTAL_ROBOTS 2
#define TOTAL_BACTERIA 10
#define EDGES 10
#define X_OFFSET 100
#define Y_OFFSET 100
#define BOUNDING_CIRCLE_RADIUS 5
#define DOUBLE_WIDTH 2
#define STEP_LENGTH 1
#define	LIFE_TIME 5
#define DIRECTIONS_NO 8
#define TERMINATE_ITER 2000
#define REPEAT_ITER 500
#define SWARMING_LIFE 2
#define REPRODUCERS_NO 0.3
#define OFFSPRINGS_NO 0.4

struct COORDINATE
{
	float x;
	float y;
};

struct LINE
{
	float a;
	float m;
	float c;
};
```

```
struct VERTEX
{
	float x;
	float y;
	bool link[TOTAL_OBSTACLES][TOTAL_VERTEX];
	COORDINATE cluster_centroid;
};

struct OBSTACLE
{
	int vertex_nos;
	VERTEX point[TOTAL_VERTEX];
	LINE line[TOTAL_VERTEX];
};

struct CPOSITION
{
	int obstacle;
	int vertex;
};

struct PARTICLE
{
	CPOSITION position[TOTAL_OBSTACLES*TOTAL_VERTEX];
	int length;
	double fitness;
};

enum DIRECTION
{
	N,NE,E,SE,S,SW,W,NW,NIL
};

struct BACTERIA_PTS
{
	float x;
	float y;
	DIRECTION dir;
};

struct PATH_SEGMENTS
{
```

```
        BACTERIA_PTS ctrl_pts[TOTAL_CONTROL_PTS];
};

struct BACTERIUM
{
        PATH_SEGMENTS segment[TOTAL_OBSTACLES*TOTAL_VERTEX];
        int segments_no;
        float fitness;
};

#endif
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: Hello.h
Intent: MFC Classes
*****************************************************************/


class CMyApp : public CWinApp
{
public:
   virtual BOOL InitInstance ();
};

class CMainWindow : public CFrameWnd
{
public:
   CMainWindow ();

protected:
   afx_msg void OnPaint ();
   DECLARE_MESSAGE_MAP ()
};
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: Hello.cpp
Intent: MFC Application Initializer
*****************************************************************/

#include <afxwin.h>
#include "Hello.h"
#include <iostream>
using namespace std;

#include "MAP.h"
#include "PSO.h"
#include "DRAW.h"
#include "BFA.h"

MAP map;
PSO pso;
BFA bfa;
CMyApp myApp;


BOOL CMyApp::InitInstance ()
{
   m_pMainWnd = new CMainWindow;
   m_pMainWnd->ShowWindow (SW_SHOWMAXIMIZED);
   m_pMainWnd->UpdateWindow ();
   return TRUE;
}

BEGIN_MESSAGE_MAP (CMainWindow, CFrameWnd)
   ON_WM_PAINT ()
END_MESSAGE_MAP ()

CMainWindow::CMainWindow ()
{
   Create (NULL, _T ("Robo
Navigator"),WS_OVERLAPPEDWINDOW|WS_VSCROLL);
}

void CMainWindow::OnPaint ()
{
        DRAW my_app;
```

```
        int k;

    CPaintDC dc (this);
        map.map_data();
        pso.pso_entry();
        pso.pso_start();
        bfa.bfa_initialize();
        bfa.bfa_life_cycle();
        my_app.draw_boundary(&dc);
        my_app.draw_obstacles(&dc);
        for(k=0;k<TOTAL_ROBOTS;k++)
        {
                my_app.draw_pso_path(&dc,k);
                my_app.draw_bfa_path(&dc,k);
                //my_app.draw_control_pts(&dc,k);
                my_app.draw_source_goal(&dc,k);
        }

    //CRect rect;
    //GetClientRect (&rect);

    //dc.DrawText (_T ("Hello, MFC"), -1, &rect,
       //DT_SINGLELINE | DT_CENTER | DT_VCENTER);
}

/*void CMainFrame::OnSysCommand(UINT nID, LPARAM lParam)
{
    UINT cmd = nID & 0xFFF0;
    if(cmd == SC_RESTORE || cmd == SC_MOVE)
       return;

    CFrameWnd::OnSysCommand(nID, lParam);
}*/
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
Intent: Map information Class
*****************************************************************/

#ifndef _MAP_H
#define _MAP_H
#pragma once

#include "STRUCTURES.h"

class MAP
{
public:
	MAP();
	void map_data();
	void map_entry();
	void map_cluster();
	void map_cluster_centroid();

	int obstacle_nos;
	OBSTACLE obstacle[TOTAL_OBSTACLES];
	COORDINATE limits[4];
};

#endif
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: MAP.cpp
Intent: Map information
*****************************************************************/

#include <iostream>
#include <fstream>
using namespace std;

#include "MAP.h"
#include "STRUCTURES.h"
#include "GEOMETRY.h"

extern MAP map;

MAP :: MAP()
{
	int i,j,m,n;
	obstacle_nos=0;

	for(i=0;i<TOTAL_OBSTACLES;i++)
	{
		obstacle[i].vertex_nos=0;
		for(j=0;j<TOTAL_VERTEX;j++)
		{
			obstacle[i].point[j].x=0;
			obstacle[i].point[j].y=0;
			for(m=0;m<TOTAL_OBSTACLES;m++)
			{
				for(n=0;n<TOTAL_VERTEX;n++)
				{
					obstacle[i].point[j].link[m][n]=false;
				}
			}
			obstacle[i].line[j].a=0;
			obstacle[i].line[j].m=0;
			obstacle[i].line[j].c=0;
		}
	}

	for(i=0;i<4;i++)
	{
```

```
                limits[i].x=0;
                limits[i].y=0;
        }
}

void MAP :: map_data()
{
        map.map_entry();
        map.map_cluster();
        map.map_cluster_centroid();
}

void MAP :: map_entry()
{
        ifstream fp;
        int i,j,k;

        cout<<"Reading map information from text file...\n";
        fp.open("map.txt",ios::in);
        if(!fp)
        {
                cout<<"FATAL ERROR : UNABLE TO READ FROM INPUT FILE
MAP.TXT";
                exit(0);
        }
        fp>>obstacle_nos;
        //fscanf(fp,"%d",&obstacle_nos);
        for(i=0;i<obstacle_nos;i++)
        {
                fp>>obstacle[i].vertex_nos;
                for(j=0;j<obstacle[i].vertex_nos;j++)
                {
                        fp>>obstacle[i].point[j].x>>obstacle[i].point[j].y;
                }
                for(j=0;j<obstacle[i].vertex_nos;j++)
                {
                        if(j+1==obstacle[i].vertex_nos)
                                k=0;
                        else
                                k=j+1;
                        obstacle[i].line[j]=line_eqn(i,j,i,k);
                }
        }
```

```
	for(i=0;i<4;i++)
	{
		fp>>limits[i].x>>limits[i].y;
	}
}

void MAP :: map_cluster()
{
	int i,j,m,n;
	for(i=0;i<obstacle_nos;i++)
	{
		for(j=0;j<obstacle[i].vertex_nos;j++)
		{
			if(j+1==obstacle[i].vertex_nos)
				obstacle[i].point[j].link[i][0]=true;
			else
				obstacle[i].point[j].link[i][j+1]=true;
			if(j-1<0)
				obstacle[i].point[j].link[i][obstacle[i].vertex_nos-1]=true;
			else
				obstacle[i].point[j].link[i][j-1]=true;

			for(m=0;m<obstacle_nos;m++)
			{
				if(m==i)	continue;
				for(n=0;n<obstacle[m].vertex_nos;n++)
				{
					if(!intersect(i,j,m,n))
						obstacle[i].point[j].link[m][n]=true;
				}
			}
		}
	}
}

void MAP :: map_cluster_centroid()
{
	int i,j,m,n,no;
	float sum_x,sum_y;

	for(i=0;i<map.obstacle_nos;i++)
	{
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
```

```
		{
			sum_x=0;
			sum_y=0;
			no=0;
			for(m=0;m<map.obstacle_nos;m++)
			{
				for(n=0;n<map.obstacle[m].vertex_nos;n++)
				{
					if(map.obstacle[i].point[j].link[m][n])
					{
						sum_x+=map.obstacle[m].point[n].x;
						sum_y+=map.obstacle[m].point[n].y;
						no++;
					}
				}
			}
			map.obstacle[i].point[j].cluster_centroid.x=sum_x/no;
			map.obstacle[i].point[j].cluster_centroid.y=sum_y/no;
		}
	}
}
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: PSO.h
Intent: Particle Swarm Optimization Class
*****************************************************************/

#ifndef _PSO_H
#define _PSO_H
#pragma once

#include "STRUCTURES.h"

class PSO
{
public:
	PSO();
	void pso_entry();
	void pso_start();
	void pso_mutate(PARTICLE&);
	bool pso_propagate(PARTICLE&,int);
	double pso_objective(VERTEX&,VERTEX&);
	bool pso_initialize(PARTICLE&,int);
	double pso_degenerate_obj(VERTEX&,VERTEX&,int);
	bool pso_repetition(PARTICLE&,int,int);
	void print_path(PARTICLE&);
	void pso_fitness_calc();

	PARTICLE particle[TOTAL_ROBOTS][TOTAL_PARTICLES];
	PARTICLE gbest[TOTAL_ROBOTS];
	bool path_complete[TOTAL_ROBOTS];
	VERTEX source[TOTAL_ROBOTS],goal[TOTAL_ROBOTS];
};
#endif
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: PSO.cpp
Intent: Particle Swarm Optimization
*****************************************************************/

#include <iostream>
#include <fstream>
#include <math.h>
#include<windows.h>
using namespace std;
//#using <mscorlib.dll>
//using namespace System;
//#include<msclr\marshal.h>
//using namespace msclr::interop;

#include "MAP.h"
#include "PSO.h"
#include "GEOMETRY.h"
//#include "STRUCTURES.h"

extern MAP map;
//DWORD lpdwProcessList,dwProcessCount;

PSO :: PSO()
{
	int i,j,k;

	for(k=0;k<TOTAL_ROBOTS;k++)
	{
		for(i=0;i<TOTAL_PARTICLES;i++)
		{
			for(j=0;j<TOTAL_OBSTACLES*TOTAL_VERTEX;j++)
			{
				particle[k][i].position[j].obstacle=-1;
				particle[k][i].position[j].vertex=-1;
			}
			particle[k][i].length=0;
			particle[k][i].fitness=0;
		}
		for(i=0;i<TOTAL_OBSTACLES*TOTAL_VERTEX;i++)
		{
			gbest[k].position[i].obstacle=-1;
```

```
                        gbest[k].position[i].vertex=-1;
                }
                gbest[k].length=0;
                gbest[k].fitness=0;
                path_complete[k]=false;
        }
}

void PSO ::pso_entry()
{
        ifstream indata_source_goal;
        indata_source_goal.open("source goal.txt",ios::in);
        int i,j,k,robots_no=0;
        while(!indata_source_goal.eof())
        {

        indata_source_goal>>source[robots_no].x>>source[robots_no].y>>goal[robots
_no].x>>goal[robots_no].y;
                robots_no++;
        }

        if(robots_no!=TOTAL_ROBOTS)
        {
                cout<<"Number of Robots and Data mismatch"<<endl;
                getchar();
        }
        /*marshal_context console_read;
        AllocConsole();
        //GetConsoleProcessList(&lpdwProcessList,dwProcessCount);
        Console::Write("\nEnter the source and the goal point\n");
        Console::Write("Source x,y\n");
        source.x=(float)atof(console_read.marshal_as<const
char*>(Console::ReadLine()));
        source.y=(float)atof(console_read.marshal_as<const
char*>(Console::ReadLine()));
        Console::Write("Goal x,y\n");
        goal.x=(float)atof(console_read.marshal_as<const
char*>(Console::ReadLine()));
        goal.y=(float)atof(console_read.marshal_as<const
char*>(Console::ReadLine()));*/
        /*cout<<"\nEnter the source and the goal point\n";
        cout<<"Source x,y\n";
        cin>>source.x>>source.y;
```

```
	cout<<"Goal x,y\n";
	cin>>goal.x>>goal.y;*/
	for(k=0;k<robots_no;k++)
	{
		if(inside_polygon_check(source[k].x,source[k].y) ||
	inside_polygon_check(goal[k].x,goal[k].y))
		{
			cout<<"Invalid Source Goal point(inside obstacles)"<<endl;
			getchar();
		}
	}
	for(i=0;i<map.obstacle_nos;i++)
	{
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
		{
			for(k=0;k<robots_no;k++)
			{
				source[k].link[i][j]=!intersect(source[k].x,source[k].y,i,j);
				goal[k].link[i][j]=!intersect(goal[k].x,goal[k].y,i,j);
			}
		}
	}
	//FreeConsole();
	indata_source_goal.close();
}

void PSO :: pso_start()
{
	int i,j,k,counter=0;
	//AllocConsole();
	//AttachConsole(lpdwProcessList);
	for(j=0;j<TOTAL_ROBOTS;j++)
	{
		if(!intersect(source[j].x,source[j].y,goal[j].x,goal[j].y))
		{
			path_complete[j]=true;
			//cout<<"\nSource Goal Straight Line\n";
			//getchar();
			//FreeConsole();
			//return;
		}
	}
```

```
for(j=0;j<TOTAL_ROBOTS;j++)
{
	for(i=0;i<TOTAL_PARTICLES;i++)
	{
		if(path_complete[j])
			continue;

		if(pso_initialize(particle[j][i],j))
		{
			//AllocConsole();
			//cout<<"\nPath found";
			//getchar();
			//FreeConsole();
			gbest[j]=particle[j][i];
			print_path(particle[j][i]);
			path_complete[j]=true;
			//return;
		}
	}
}

for(k=0;k<TOTAL_ROBOTS;k++)
{
	if(path_complete[k])
		continue;

	while(1)
	{
		counter++;
		if(counter>TERMINATE_ITER)
		{
			//cout<<"\nUnable to find path";
			counter=0;
			//getchar();
			//FreeConsole();
			//return;
			break;
		}
		for(i=0;i<TOTAL_PARTICLES;i++)
		{
			if(pso_propagate(particle[k][i],k))
			{
				//cout<<"\nPath found";
```

```
                                        //getchar();
                                        //FreeConsole();
                                        gbest[k]=particle[k][i];
                                        print_path(particle[k][i]);
                                        path_complete[k]=true;
                                        //return;
                                }
                        }
                        if(path_complete[k])
                                break;
                }
        }
        pso_fitness_calc();
}

void PSO :: pso_mutate(PARTICLE& ptcle)
{

}

bool PSO :: pso_propagate(PARTICLE& ptcle, int k)
{
        int obs,vert,obs1,vert1,i,j,init=0;
        double obj,temp_obj;
        bool rep;
        CPOSITION temp;
        bool propagate=false;

        obs=ptcle.position[ptcle.length-1].obstacle;
        vert=ptcle.position[ptcle.length-1].vertex;

        for(i=0;i<map.obstacle_nos;i++)
        {
                for(j=0;j<map.obstacle[i].vertex_nos;j++)
                {
                        if(map.obstacle[obs].point[vert].link[i][j])
                        {
                                temp_obj=pso_objective(map.obstacle[i].point[j],goal[k]);
                                rep=pso_repetition(ptcle,i,j);
                                if(rep)                 continue;
                                if(init==0)
                                {
                                        obj=temp_obj;
```

```
                                temp.obstacle=i;
                                temp.vertex=j;
                                init=1;
                                propagate=true;
                                continue;
                        }
                        if(temp_obj<obj)
                        {
                                //rand_counter=rand()/RAND_MAX;
                                //if(rand_counter<0.2)        continue;
                                obj=temp_obj;
                                temp.obstacle=i;
                                temp.vertex=j;
                        }
                }
        }
}
if(!propagate)
{
        for(i=0;i<map.obstacle_nos;i++)
        {
                for(j=0;j<map.obstacle[i].vertex_nos;j++)
                {
                        if(map.obstacle[obs].point[vert].link[i][j])
                        {

temp_obj=pso_degenerate_obj(map.obstacle[obs].point[vert],map.obstacle[i].p
oint[j],k);
                                if(temp_obj<0)        continue;
                                if(init==0)
                                {
                                        obj=temp_obj;
                                        temp.obstacle=i;
                                        temp.vertex=j;
                                        init=1;
                                        propagate=true;
                                        continue;
                                }
                                if(temp_obj<obj)
                                {
                                        //rand_counter=rand()/RAND_MAX;
                                        //if(rand_counter<0.7)        continue;
                                        obj=temp_obj;
```

```
                                        temp.obstacle=i;
                                        temp.vertex=j;
                                    }
                                }
                            }
                        }
                }
        if(!propagate)
        {
                ptcle.position[ptcle.length-1].obstacle=0;
                ptcle.position[ptcle.length-1].vertex=0;
                ptcle.length--;
                obs1=ptcle.position[ptcle.length-2].obstacle;
                vert1=ptcle.position[ptcle.length-2].vertex;
                map.obstacle[obs1].point[vert1].link[obs][vert]=false;
                return false;
        }
        ptcle.position[ptcle.length]=temp;
        ptcle.length++;

        if(goal[k].link[ptcle.position[ptcle.length-
1].obstacle][ptcle.position[ptcle.length-1].vertex])
                return true;
        else
                return false;

        /*if(obj==0)
                return true;
        else
                return false;*/
}

double PSO :: pso_objective(VERTEX& pt1,VERTEX& pt2)
{
        double x_diff,y_diff;
        x_diff=pt1.x-pt2.x;
        y_diff=pt1.y-pt2.y;
        return(pow((pow(x_diff,2)+pow(y_diff,2)),0.5));
}

double PSO :: pso_degenerate_obj(VERTEX& pt1,VERTEX& pt2,int k)
{
```

```
	if((fabs(pt1.x-goal[k].x)<=fabs(pt2.x-goal[k].x))^(fabs(pt1.y-
goal[k].y)<=fabs(pt2.y-goal[k].y)))
	{
		if(fabs(pt1.x-goal[k].x)<=fabs(pt2.x-goal[k].x))		return
(fabs(pt2.x-goal[k].x));
		if(fabs(pt1.y-goal[k].y)<=fabs(pt2.y-goal[k].y))		return
(fabs(pt2.y-goal[k].y));
	}
	return(-1);
}

bool PSO :: pso_initialize(PARTICLE& ptcle, int k)
{
	int i,j,init=0;
	double obj,temp_obj;
	CPOSITION temp;
	bool propagate=false;

	for(i=0;i<map.obstacle_nos;i++)
	{
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
		{
			if(source[k].link[i][j])
			{
				temp_obj=pso_objective(map.obstacle[i].point[j],goal[k]);
				if(init==0)
				{
					obj=temp_obj;
					temp.obstacle=i;
					temp.vertex=j;
					init=1;
					propagate=true;
					continue;
				}
				if(temp_obj<obj)
				{
					//rand_counter=rand()/RAND_MAX;
					//if(rand_counter<0.2)	continue;
					obj=temp_obj;
					temp.obstacle=i;
					temp.vertex=j;
				}
			}
```

```
                }
        }

        ptcle.position[ptcle.length]=temp;
        ptcle.length++;

        if(goal[k].link[ptcle.position[ptcle.length-
1].obstacle][ptcle.position[ptcle.length-1].vertex])
                return true;
        else
                return false;
}

bool PSO :: pso_repetition(PARTICLE& ptcle,int p, int q)
{
        int i;
        for(i=ptcle.length-1;i>=0;i--)
        {
                if(ptcle.position[i].obstacle==p && ptcle.position[i].vertex==q)
                        return true;
        }
        return false;
}

void PSO :: pso_fitness_calc()
{
        int i,k;

        for(k=0;k<TOTAL_ROBOTS;k++)
        {
                        gbest[k].fitness=0;
                        if(gbest[k].length==0 && path_complete[k])
                        {
                                gbest[k].fitness+=pso_objective(source[k],goal[k]);
                        }

        gbest[k].fitness+=pso_objective(source[k],map.obstacle[gbest[k].position[0].ob
stacle].point[gbest[k].position[0].vertex]);
                        for(i=0;i<gbest[k].length-1;i++)
                        {

        gbest[k].fitness+=pso_objective(map.obstacle[gbest[k].position[i].obstacle].poi
nt[gbest[k].position[i].vertex],
```

```
map.obstacle[gbest[k].position[i].obstacle].point[gbest[k].position[i].vertex]);
                        }

        gbest[k].fitness+=pso_objective(map.obstacle[gbest[k].position[gbest[k].length
-1].obstacle].point[gbest[k].position[gbest[k].length-1].vertex],goal[k]);
        }
}

void PSO :: print_path(PARTICLE& ptcle)
{
        /*int i;
        AttachConsole(lpdwProcessList);
        AllocConsole();
        cout<<"\nPath"<<endl;
        for(i=0;i<gbest.length;i++)
        {
                cout<<gbest.position[i].obstacle<<"\t"<<gbest.position[i].vertex<<endl;
        }
        getchar();
        FreeConsole();*/
}
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: Geometry.h
Intent: Geometry of obstacles
*****************************************************************/

#ifndef _GEOMETRY_H
#define _GEOMETRY_H
#pragma once

#include "STRUCTURES.h"

LINE line_eqn(int obs1,int pt1,int obs2,int pt2);
LINE line_eqn(float ptx,float pty,int obs2,int pt2);
LINE line_eqn(float ptx1,float pty1,float ptx2,float pty2);
bool intersect(int obs1,int pt1,int obs2,int pt2);
bool intersect(float ptx,float pty,int obs2,int pt2);
bool intersect(float ptx1,float pty1,float ptx2,float pty2);
bool intersect_check(float x1,float y1,float x2,float y2,float x3, float y3, float x4,float
y4, float ix, float iy);
bool inside_polygon_check(float,float);

#endif
```

```
/*********************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: GEOMETRY.cpp
Intent: Geometry of obstacles
*********************************************************************/

#include <iostream>
#include <math.h>
using namespace std;

#include "GEOMETRY.h"
#include "STRUCTURES.h"
#include "MAP.h"

extern MAP map;

LINE line_eqn(int obs1,int pt1,int obs2,int pt2)
{
	LINE line;
	float x_diff,y_diff;

	x_diff=map.obstacle[obs1].point[pt1].x-map.obstacle[obs2].point[pt2].x;
	y_diff=map.obstacle[obs1].point[pt1].y-map.obstacle[obs2].point[pt2].y;

	if(fabs(x_diff)<ZERO)
	{
		line.a=0;
		line.m=1;
		line.c=-map.obstacle[obs1].point[pt1].x;
	}
	else
	{
		line.a=1;
		line.m=y_diff/x_diff;
		if(fabs(line.m)<ZERO)
			line.m=0;

		line.c=map.obstacle[obs1].point[pt1].y-
line.m*map.obstacle[obs1].point[pt1].x;
	}
	return(line);
}
```

```
LINE line_eqn(float ptx,float pty,int obs2,int pt2)
{
	LINE line;
	float x_diff,y_diff;

	x_diff=ptx-map.obstacle[obs2].point[pt2].x;
	y_diff=pty-map.obstacle[obs2].point[pt2].y;

	if(fabs(x_diff)<ZERO)
	{
		line.a=0;
		line.m=1;
		line.c=-ptx;
	}
	else
	{
		line.a=1;
		line.m=y_diff/x_diff;
		if(fabs(line.m)<ZERO)
			line.m=0;

		line.c=map.obstacle[obs2].point[pt2].y-
line.m*map.obstacle[obs2].point[pt2].x;
	}
	return(line);
}

LINE line_eqn(float ptx1,float pty1,float ptx2,float pty2)
{
	LINE line;
	float x_diff,y_diff;

	x_diff=ptx1-ptx2;
	y_diff=pty1-pty2;

	if(fabs(x_diff)<ZERO)
	{
		line.a=0;
		line.m=1;
		line.c=-ptx1;
	}
	else
	{
```

```
		line.a=1;
		line.m=y_diff/x_diff;
		if(fabs(line.m)<ZERO)
			line.m=0;

		line.c=pty2-line.m*ptx2;
	}
	return(line);
}

bool intersect(int obs1,int pt1,int obs2,int pt2)
{
	LINE line;
	int i,j,k,p,q;
	float delta,delta_x,delta_y;
	float intersect_x,intersect_y;

	line=line_eqn(obs1,pt1,obs2,pt2);

	for(i=0;i<map.obstacle_nos;i++)
	{
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
		{
			delta=line.a*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].a*line.m;
			delta_x=line.c*map.obstacle[i].line[j].a-
map.obstacle[i].line[j].c*line.a;
			delta_y=line.c*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].c*line.m;

			if(fabs(delta)<ZERO)	continue;

			if(j+1==map.obstacle[i].vertex_nos)
				k=0;
			else
				k=j+1;

			intersect_x=delta_x/delta;
			intersect_y=delta_y/delta;

			/*if(
				((((map.obstacle[obs1].point[pt1].x-
intersect_x>zero) && (map.obstacle[obs2].point[pt2].x-intersect_x<zero))||
```

```
                                        ((map.obstacle[obs1].point[pt1].x-
intersect_x<zero) && (map.obstacle[obs2].point[pt2].x-intersect_x>zero)))&&
                                        (((map.obstacle[obs1].point[pt1].y-
intersect_y>zero) && (map.obstacle[obs2].point[pt2].y-intersect_y<zero))||
                                        ((map.obstacle[obs1].point[pt1].y-
intersect_y<zero) && (map.obstacle[obs2].point[pt2].y-intersect_y>zero))))

                                                ||
                                        ((((map.obstacle[i].point[j].x-intersect_x>zero) &&
(map.obstacle[i].point[k].x-intersect_x<zero))||
                                        ((map.obstacle[i].point[j].x-intersect_x<zero) &&
(map.obstacle[i].point[k].x-intersect_x>zero)))&&
                                        (((map.obstacle[i].point[j].y-intersect_y>zero) &&
(map.obstacle[i].point[k].y-intersect_y<zero))||
                                        ((map.obstacle[i].point[j].y-intersect_y<zero) &&
(map.obstacle[i].point[k].y-intersect_y>zero))))
                                )*/

                                        for(p=0;p<map.obstacle_nos;p++)
                                        {
                                                for(q=0;q<map.obstacle[p].vertex_nos;q++)
                                                {
                                                        if((p==obs1 && q==pt1) || (p==obs2 &&
q==pt2))
                                                        {
                                                                if(fabs(intersect_x-
map.obstacle[p].point[q].x)<ZERO && fabs(intersect_y-
map.obstacle[p].point[q].y)<ZERO)
                                                                        goto chk_end;
                                                        }
                                                        if(fabs(intersect_x-
map.obstacle[p].point[q].x)<ZERO && fabs(intersect_y-
map.obstacle[p].point[q].y)<ZERO)
                                                        {

        if(intersect_check(map.obstacle[obs1].point[pt1].x,map.obstacle[obs1].point[pt
1].y,

map.obstacle[obs2].point[pt2].x,map.obstacle[obs2].point[pt2].y,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
```

```
                                                        intersect_x,intersect_y))
                                                                return(true);
                                                }
                                        }
                                }


        if(intersect_check(map.obstacle[obs1].point[pt1].x,map.obstacle[obs1].point[pt
1].y,

map.obstacle[obs2].point[pt2].x,map.obstacle[obs2].point[pt2].y,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
                                                        intersect_x,intersect_y))
                                return(true);
chk_end:;
                }
        }
        return(false);
}

bool intersect(float ptx,float pty,int obs2,int pt2)
{
        LINE line;
        int i,j,k,p,q;
        float delta,delta_x,delta_y;
        float intersect_x,intersect_y;

        line=line_eqn(ptx,pty,obs2,pt2);

        for(i=0;i<map.obstacle_nos;i++)
        {
                for(j=0;j<map.obstacle[i].vertex_nos;j++)
                {
                        delta=line.a*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].a*line.m;
                        delta_x=line.c*map.obstacle[i].line[j].a-
map.obstacle[i].line[j].c*line.a;
                        delta_y=line.c*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].c*line.m;
```

```
if(fabs(delta)<ZERO)        continue;

if(j+1==map.obstacle[i].vertex_nos)
        k=0;
else
        k=j+1;

intersect_x=delta_x/delta;
intersect_y=delta_y/delta;

/*if(
                ((((ptx-intersect_x>ZERO) &&
(map.obstacle[obs2].point[pt2].x-intersect_x<ZERO))||
                 ((ptx-intersect_x<ZERO) &&
(map.obstacle[obs2].point[pt2].x-intersect_x>ZERO)))&&
                (((pty-intersect_y>ZERO) &&
(map.obstacle[obs2].point[pt2].y-intersect_y<ZERO))||
                 ((pty-intersect_y<ZERO) &&
(map.obstacle[obs2].point[pt2].y-intersect_y>ZERO))))

                        &&
                ((((map.obstacle[i].point[j].x-intersect_x>ZERO)
&& (map.obstacle[i].point[k].x-intersect_x<ZERO))||
                 ((map.obstacle[i].point[j].x-intersect_x<ZERO)
&& (map.obstacle[i].point[k].x-intersect_x>ZERO)))&&
                (((map.obstacle[i].point[j].y-intersect_y>ZERO)
&& (map.obstacle[i].point[k].y-intersect_y<ZERO))||
                 ((map.obstacle[i].point[j].y-intersect_y<ZERO)
&& (map.obstacle[i].point[k].y-intersect_y>ZERO))))
         )*/

        for(p=0;p<map.obstacle_nos;p++)
        {
                for(q=0;q<map.obstacle[p].vertex_nos;q++)
                {
                        if(p==obs2 && q==pt2)
                        {
                                if(fabs(intersect_x-
map.obstacle[p].point[q].x)<ZERO && fabs(intersect_y-
map.obstacle[p].point[q].y)<ZERO)
                                        goto chk_end;
                        }
```

```
                                        if(intersect_x==map.obstacle[p].point[q].x
&& intersect_y==map.obstacle[p].point[q].y)
                                        {
                                                if(intersect_check(ptx,pty,

map.obstacle[obs2].point[pt2].x,map.obstacle[obs2].point[pt2].y,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
                                                        intersect_x,intersect_y))
                                                                return(true);
                                        }
                                }
                        }

                if(intersect_check(ptx,pty,

map.obstacle[obs2].point[pt2].x,map.obstacle[obs2].point[pt2].y,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
                                                        intersect_x,intersect_y))
                        return(true);
chk_end:;
                }
        }
        return(false);
}

bool intersect(float ptx1,float pty1,float ptx2,float pty2)
{
        LINE line;
        int i,j,k,p,q;
        float delta,delta_x,delta_y;
        float intersect_x,intersect_y;

        line=line_eqn(ptx1,pty1,ptx2,pty2);

        for(i=0;i<map.obstacle_nos;i++)
        {
                for(j=0;j<map.obstacle[i].vertex_nos;j++)
```

```
		{
			delta=line.a*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].a*line.m;
			delta_x=line.c*map.obstacle[i].line[j].a-
map.obstacle[i].line[j].c*line.a;
			delta_y=line.c*map.obstacle[i].line[j].m-
map.obstacle[i].line[j].c*line.m;

			if(fabs(delta)<ZERO)	continue;

			if(j+1==map.obstacle[i].vertex_nos)
				k=0;
			else
				k=j+1;

			intersect_x=delta_x/delta;
			intersect_y=delta_y/delta;

			/*if(
					((((ptx1-intersect_x>ZERO) && (ptx2-
intersect_x<ZERO))||
					 ((ptx1-intersect_x<ZERO) && (ptx2-
intersect_x>ZERO)))&&
					(((pty1-intersect_y>ZERO) && (pty2-
intersect_y<ZERO))||
					 ((pty1-intersect_y<ZERO) && (pty2-
intersect_y>ZERO))))

						&&
					((((map.obstacle[i].point[j].x-intersect_x>ZERO)
&& (map.obstacle[i].point[k].x-intersect_x<ZERO))||
					 ((map.obstacle[i].point[j].x-intersect_x<ZERO)
&& (map.obstacle[i].point[k].x-intersect_x>ZERO)))&&
					(((map.obstacle[i].point[j].y-intersect_y>ZERO)
&& (map.obstacle[i].point[k].y-intersect_y<ZERO))||
					 ((map.obstacle[i].point[j].y-intersect_y<ZERO)
&& (map.obstacle[i].point[k].y-intersect_y>ZERO))))
				)*/

				for(p=0;p<map.obstacle_nos;p++)
				{
					for(q=0;q<map.obstacle[p].vertex_nos;q++)
					{
```

```
                                        if(intersect_x==map.obstacle[p].point[q].x
&& intersect_y==map.obstacle[p].point[q].y)
                                        {
                                                if(intersect_check(ptx1,pty1,
                                                  ptx2,pty2,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
                                                  intersect_x,intersect_y))
                                                        return(true);
                                        }
                                }
                        }

                if(intersect_check(ptx1,pty1,
                                                  ptx2,pty2,

map.obstacle[i].point[j].x,map.obstacle[i].point[j].y,

map.obstacle[i].point[k].x,map.obstacle[i].point[k].y,
                                                  intersect_x,intersect_y))
                                return(true);
                        }
        }
        return(false);
}

bool intersect_check(float x1,float y1,float x2,float y2,float x3, float y3, float x4,float
y4, float ix, float iy)
{
        float greater_x,greater_y,smaller_x,smaller_y;

        if(x1>x2)
        {
                greater_x=x1;
                smaller_x=x2;
        }
        else
        {
                greater_x=x2;
                smaller_x=x1;
        }
```

```
	if(y1>y2)
	{
		greater_y=y1;
		smaller_y=y2;
	}
	else
	{
		greater_y=y2;
		smaller_y=y1;
	}

	//if(!((greater_x-ix>-ZERO && ix-smaller_x>-ZERO)&&(greater_y-iy>-
ZERO && iy-smaller_y>-ZERO)))
	if(!((greater_x>=ix && ix>=smaller_x)&&(greater_y>=iy &&
iy>=smaller_y)))
	{
		return(false);
	}
	/*else
	{
		return(true);
	}*/

	if(x3>x4)
	{
		greater_x=x3;
		smaller_x=x4;
	}
	else
	{
		greater_x=x4;
		smaller_x=x3;
	}
	if(y3>y4)
	{
		greater_y=y3;
		smaller_y=y4;
	}
	else
	{
		greater_y=y4;
		smaller_y=y3;
	}
```

```
	//if(!(greater_x-ix>-ZERO && ix-smaller_x>-ZERO)&&(greater_y-iy>-ZERO
&& iy-smaller_y>-ZERO))
	if(!((greater_x>=ix && ix>=smaller_x)&&(greater_y>=iy &&
iy>=smaller_y)))
	{
		return(false);
	}
	/*else
	{
		return(true);
	}*/

	return(true);
}

bool inside_polygon_check(float x, float y)
{
	int i,j,k;
	float ptx1,pty1,ptx2,pty2;
	bool horizontal1,horizontal2;
	bool vertical1,vertical2;

	for(i=0;i<map.obstacle_nos;i++)
	{
		horizontal1=false;
		horizontal2=false;
		vertical1=false;
		vertical2=false;
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
		{
			if(j+1==map.obstacle[i].vertex_nos)
				k=0;
			else
				k=j+1;
			if(map.obstacle[i].point[j].x>map.obstacle[i].point[k].x)
			{
				ptx1=map.obstacle[i].point[j].x;
				ptx2=map.obstacle[i].point[k].x;
			}
			else
			{
				ptx1=map.obstacle[i].point[k].x;
```

```
                        ptx2=map.obstacle[i].point[j].x;
                }
                if(map.obstacle[i].point[j].y>map.obstacle[i].point[k].y)
                {
                        pty1=map.obstacle[i].point[j].y;
                        pty2=map.obstacle[i].point[k].y;
                }
                else
                {
                        pty1=map.obstacle[i].point[k].y;
                        pty2=map.obstacle[i].point[j].y;
                }

                if(!(map.obstacle[i].line[j].m==0))
                {
                        if((y<pty1 && y>pty2) && (fabs(y-pty1)>ZERO &&
fabs(y-pty2)>ZERO))
                        {
                                if(horizontal1)
                                        horizontal2=true;
                                else
                                        horizontal1=true;
                        }
                }
                if(!(map.obstacle[i].line[j].a==0))
                {
                        if((x<ptx1 && x>ptx2) && (fabs(x-ptx1)>ZERO &&
fabs(x-ptx2)>ZERO))
                        {
                                if(vertical1)
                                        vertical2=true;
                                else
                                        vertical1=true;
                        }
                }
        }
        if(horizontal1 && horizontal2 && vertical1 && vertical2)
                return(true);
    }
    return(false);
}
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: DRAW.h
Intent: Windows Graphics Classes
*****************************************************************/

#ifndef _DRAW_H
#define _DRAW_H
#pragma once

#include <windows.h>
#include "STRUCTURES.h"

class DRAW
{
public:
	void draw_boundary(CPaintDC*);
	void draw_obstacles(CPaintDC*);
	void draw_pso_path(CPaintDC*,int);
	void draw_bfa_path(CPaintDC*,int);
	void initialize_pts(CPoint&);
	void draw_source_goal(CPaintDC*,int);
	void draw_control_pts(CPaintDC*,int);
	void source_goal_pts(CPoint&,CPoint&,int);
	void ctrl_pts_init(CPoint*,int);
	CRect bounding_box(CPoint&);
	void initialize_rect(CRect*);
};

#endif
```

```
/*********************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: DRAW.cpp
Intent: Graphics Functions
*********************************************************************/

#include<iostream>
#include<math.h>
#include<afxwin.h>
using namespace std;

#include "DRAW.h"
#include "PSO.h"
#include "MAP.h"
#include "BFA.h"
#include "STRUCTURES.h"

extern MAP map;
extern PSO pso;
extern BFA bfa;

COLORREF BLACK=RGB(0,0,0);
COLORREF BLUE=RGB(0,0,192);
COLORREF GREEN=RGB(0,255,0);
COLORREF RED=RGB(255,0,0);
COLORREF ORANGE=RGB(255,140,0);
COLORREF INDIAN_RED=RGB(205,92,92);

CBrush BLUE_BRUSH(BLUE);
CBrush RED_BRUSH(RED);
CBrush GREEN_BRUSH(GREEN);
CBrush ORANGE_BRUSH(ORANGE);
CBrush INDIAN_RED_BRUSH(INDIAN_RED);

CPen BLUE_PEN(PS_SOLID,DOUBLE_WIDTH,BLUE);
CPen RED_PEN(PS_SOLID,DOUBLE_WIDTH,RED);
CPen GREEN_PEN(PS_SOLID,DOUBLE_WIDTH,GREEN);
CPen ORANGE_PEN(PS_SOLID,DOUBLE_WIDTH,ORANGE);
CPen INDIAN_RED_PEN(PS_SOLID,DOUBLE_WIDTH,INDIAN_RED);
CPen nBLACK_PEN(PS_SOLID,DOUBLE_WIDTH,BLACK);
CPen nBLACK_PEN_THICK(PS_SOLID,DOUBLE_WIDTH*2,BLACK);
CPen
INDIAN_RED_PEN_THICK(PS_SOLID,DOUBLE_WIDTH*2,INDIAN_RED);
```

```
void DRAW ::draw_boundary(CPaintDC* dc)
{
	int arr_size,i;
	CPoint pts[EDGES];
	arr_size=(sizeof(map.limits)/sizeof(float))/2;
	for(i=0;i<arr_size;i++)
	{
		pts[i].x=(long)map.limits[i].x;
		pts[i].y=(long)map.limits[i].y;
		pts[i].Offset(X_OFFSET,Y_OFFSET);
	}
	pts[arr_size].x=(long)map.limits[0].x;
	pts[arr_size].y=(long)map.limits[0].y;
	pts[arr_size].Offset(X_OFFSET,Y_OFFSET);
	Polyline(*dc,pts,arr_size+1);
}

void DRAW ::draw_obstacles(CPaintDC* dc)
{
	CPoint pts[EDGES];
	CRgn obs_poly;
	int i,j;
	for(i=0;i<EDGES;i++)
	{
		initialize_pts(pts[i]);
	}
	for(i=0;i<map.obstacle_nos;i++)
	{
		for(j=0;j<map.obstacle[i].vertex_nos;j++)
		{
			pts[j].x=(long)map.obstacle[i].point[j].x;
			pts[j].y=(long)map.obstacle[i].point[j].y;
			pts[j].Offset(X_OFFSET,Y_OFFSET);
		}
		dc->SelectObject(BLUE_PEN);
		dc->SelectObject(BLUE_BRUSH);
		Polygon(*dc,pts,map.obstacle[i].vertex_nos);

	/*obs_poly.CreatePolygonRgn(pts,map.obstacle[i].vertex_nos,ALTERNATE);
		obs_poly.OffsetRgn(X_OFFSET,Y_OFFSET);
		FillRgn(*dc,obs_poly,p_Solid_Brush);
		obs_poly.DeleteObject();*/
```

```
	}
}

void DRAW ::draw_pso_path(CPaintDC* dc,int robot)
{
	int i;
	dc->SelectObject(nBLACK_PEN_THICK);

	/*dc-
>MoveTo((int)map.obstacle[pso.gbest.position[0].obstacle].point[pso.gbest.position[0
].vertex].x,

(int)map.obstacle[pso.gbest.position[0].obstacle].point[pso.gbest.position[0].vertex].y
);*/

	dc-
>MoveTo((int)(pso.source[robot].x+X_OFFSET),(int)(pso.source[robot].y+Y_OFFS
ET));

	for(i=0;i<pso.gbest[robot].length;i++)
	{
		dc-
>LineTo((int)(map.obstacle[pso.gbest[robot].position[i].obstacle].point[pso.gbest[rob
ot].position[i].vertex].x+X_OFFSET),

(int)(map.obstacle[pso.gbest[robot].position[i].obstacle].point[pso.gbest[robot].positio
n[i].vertex].y+Y_OFFSET));
	}

	dc-
>LineTo((int)(pso.goal[robot].x+X_OFFSET),(int)(pso.goal[robot].y+Y_OFFSET));
}

void DRAW ::draw_bfa_path(CPaintDC* dc,int robot)
{
	int i,j;
	dc->SelectObject(INDIAN_RED_PEN_THICK);

	/*dc-
>MoveTo((int)map.obstacle[pso.gbest.position[0].obstacle].point[pso.gbest.position[0
].vertex].x,
```

```
(int)map.obstacle[pso.gbest.position[0].obstacle].point[pso.gbest.position[0].vertex].y
);*/

	dc-
>MoveTo((int)(pso.source[robot].x+X_OFFSET),(int)(pso.source[robot].y+Y_OFFS
ET));

	for(i=0;i<bfa.gbest[robot].segments_no;i++)
	{
		for(j=0;j<TOTAL_CONTROL_PTS;j++)
		{
			dc-
>LineTo((int)(bfa.gbest[robot].segment[i].ctrl_pts[j].x+X_OFFSET),

(int)(bfa.gbest[robot].segment[i].ctrl_pts[j].y+Y_OFFSET));
		}
	}
}

void DRAW::draw_source_goal(CPaintDC* dc,int robot)
{
	CPoint source,goal;

	source_goal_pts(source,goal,robot);
	//SetPixel(*dc,source.x,source.y,GREEN);
	dc->SelectObject(&GREEN_PEN);
	dc->SelectObject(&GREEN_BRUSH);
	dc->Ellipse(bounding_box(source));
	//SetPixel(*dc,goal.x,goal.y,RED);
	dc->SelectObject(&RED_PEN);
	dc->SelectObject(&RED_BRUSH);
	dc->Ellipse(bounding_box(goal));

}

void DRAW::draw_control_pts(CPaintDC* dc,int robot)
{
	CPoint
ctrl_pts[TOTAL_OBSTACLES*TOTAL_VERTEX][TOTAL_CONTROL_PTS];
	int j,k;
	dc->SelectObject(&ORANGE_PEN);
	dc->SelectObject(&ORANGE_BRUSH);
```

```
	for(j=0;j<TOTAL_OBSTACLES*TOTAL_VERTEX;j++)
	{
		for(k=0;k<TOTAL_CONTROL_PTS;k++)
		{
			ctrl_pts[j][k].x=0;
			ctrl_pts[j][k].y=0;
		}
	}

	for(j=0;j<bfa.gbest[robot].segments_no;j++)
	{
		for(k=0;k<TOTAL_CONTROL_PTS;k++)
		{
			ctrl_pts[j][k].x=(int)bfa.gbest[robot].segment[j].ctrl_pts[k].x;
			ctrl_pts[j][k].y=(int)bfa.gbest[robot].segment[j].ctrl_pts[k].y;
			dc->Ellipse(bounding_box(ctrl_pts[j][k]));
		}
	}
}

void DRAW::source_goal_pts(CPoint& source,CPoint& goal,int i)
{
	initialize_pts(source);
	initialize_pts(goal);
	source.x=(long)pso.source[i].x;
	source.y=(long)pso.source[i].y;
	goal.x=(long)pso.goal[i].x;
	goal.y=(long)pso.goal[i].y;
}

void DRAW ::ctrl_pts_init(CPoint* pts,int i)
{
	int j,k;
	for(j=0;j<TOTAL_OBSTACLES*TOTAL_VERTEX;j++)
	{
		for(k=0;k<TOTAL_CONTROL_PTS;k++)
		{
			(pts+k)->x=(int)bfa.gbest[i].segment[j].ctrl_pts[k].x;
			(pts+k)->y=(int)bfa.gbest[i].segment[j].ctrl_pts[k].y;
		}
	}
}
```

```
CRect DRAW ::bounding_box(CPoint& pt)
{
	CRect rect;
	initialize_rect(&rect);
	rect.left=pt.x-BOUNDING_CIRCLE_RADIUS;
	rect.top=pt.y-BOUNDING_CIRCLE_RADIUS;
	rect.right=pt.x+BOUNDING_CIRCLE_RADIUS;
	rect.bottom=pt.y+BOUNDING_CIRCLE_RADIUS;
	rect.OffsetRect(X_OFFSET,Y_OFFSET);
	return(rect);
}


void DRAW ::initialize_pts(CPoint& pts)
{
	pts.x=0;
	pts.y=0;
	//pts.Offset(X_OFFSET,Y_OFFSET);
}

void DRAW ::initialize_rect(CRect* rect)
{
	//rect->OffsetRect(X_OFFSET,Y_OFFSET);
	rect->left=0;
	rect->top=0;
	rect->right=0;
	rect->bottom=0;
}
```

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: map.txt
Intent: Map Data
*****************************************************************/

9
4
120.0 160.0
120.0 130.0
180.0 130.0
180.0 160.0
4
100.0 220.0
120.0 190.0
140.0 220.0
120.0 250.0
4
200.0 180.0
250.0 180.0
230.0 220.0
180.0 220.0
4
290.0 160.0
320.0 160.0
320.0 250.0
290.0 250.0
4
160.0 270.0
230.0 270.0
210.0 310.0
180.0 310.0
4
250.0 250.0
290.0 250.0
290.0 270.0
250.0 270.0
4
330.0 220.0
360.0 220.0
360.0 320.0
330.0 320.0
3
```

150.0 310.0
220.0 380.0
150.0 380.0
4
260.0 330.0
330.0 330.0
330.0 380.0
260.0 380.0

0.0 0.0
0.0 500.0
500.0 500.0
500.0 0.0

```
/*****************************************************************
Project:Multi-Robot Coordination using Swarm Intelligence and Bacteria Foraging
File: source goal.txt
Intent: Source Goal Data
*****************************************************************/

130     370
400     100
250 390
400 100
```