1-1-2007

Implementation of a novel reactive navigation algorithm

Vijay Somers
Ryerson University

IMPLEMENTATION OF A NOVEL REACTIVE NAVIGATION ALGORITHM

by

Vijay Somers, Hon. BASc
Toronto, Ontario

A project
presented to Ryerson University

in partial fulfillment of the
requirements for the degree of
Master of Engineering
in the Program of
Electrical Engineering

Toronto, Ontario, Canada, 2007

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

Vijay Somers

I further authorize Ryerson University to reproduce this project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Vijay Somers

IMPLEMENTATION OF A NOVEL REACTIVE NAVIGATION ALGORITHM

Vijay Somers

Master of Engineering Project - Electrical Engineering
Ryerson University
September 2007

Supervisor: Prof. Alex Ferworn

ABSTRACT

In this project a reactive navigation algorithm is applied to a non-holonomic differential drive robot. The algorithm uses a stochastic process to navigate a robot through terrain while lacking a priori information. A graph is made from a random array of points that is used to connect the current location of the robot to its destination. Dijkstra's algorithm is used to select the shortest route that leads to the destination. The robot attempts to traverse this route until it detects that it is being blocked by an obstacle. The graph is then recreated with different random points, and a new route is calculated. This procedure is repeated until the robot arrives at its destination. This is tested by making a simulated robot with perfect localization travel through two kinds of environments. Processing speed is maintained by hashing location information according to its coordinates.

Keywords: Reactive Navigation, Dijkstra's algorithm, Bidirectional Graph

ACKNOWLEDGEMENTS

I express sincere appreciation to Prof. Alex Ferworn for his guidance and insight throughout the research, and for his patience with my endless last minute problems.

My family has my thanks for their understanding, motivation and for giving me the opportunity to attend this institution. Lastly, but certainly not the least, I thank all my colleagues and friends who have made my time at the university a memorable and valuable experience.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF APPENDICES

APPENDIX

CHAPTER 1

INTRODUCTION

The emerging technology of autonomous robots demands increasingly sophisticated robots that can navigate themselves around their environment without the intervention of a human handler. Programs like the DARPA grand challenge have been set up in order to foster the development of new navigation and automation systems, which have further applications in both civilian and military use. This project is an attempt to implement a pathfinding algorithm that allows a robot to travel through an unmapped area, from one location to another and avoid any obstacles in the way. The robot would have no a priori information about where it is going, only the coordinates of its destination. The algorithm operates by randomly generating a graph that has the location of the robot and its destination as vertices. The robot then attempts to travel along the shortest route from one end to another, while avoiding graph edges that intersect with obstacles.

The coding was done in C++ and made use of the ActiveMedia ARIA and ARNL control libraries, as well as an implementation of Dijkstra's algorithm that was produced by the University of Canterbury[1].

In this paper we discuss the modeling of the robot in Part 2, and the design of the algorithm in Part 3. In Part 4, the results are shown. In Part 5, strengths and weaknesses of the algorithm are discussed, with the conclusions being drawn in Part 6.

CHAPTER 2

ROBOT MODELING

This algorithm was designed and tested specifically for the Pioneer 2 research robot, produced by Activemedia Robotics. The Pioneer 2 robot is non-holonomic and is driven by a two wheel differential drive with a caster wheel to provide balance. A belt of 8 sonar sensors scan the forward facing 180° of the robot[2]. The robot is capable of odometric localization, but the error accumulates to an unacceptably large size when the robot is in operation for more than a short time. Since the algorithm presented here requires a high degree of localization accuracy, all the testing was performed using an idealized simulation of the robot. This simulation had perfect odometric localization, as well as narrower sonar beams, increasing the accuracy of the obstacle localization.

More specifically, the algorithm presented here assumes that accurate localization is available, but a map of the territory being traversed is not. This is analogous to an automated vehicle traveling from one GPS coordinate to another, but not possessing a map of the intervening space. In the absence of GPS, a system similar to the SIVIA algorithm described by Martinez *et al* would also be an appropriate application for this algorithm[3-4]. In that system, localization is accomplished by orienting the robot according to a series of beacons with known locations. The position of the beacons remains static, but the territory around them can vary.

The effective radius of the robot was found to be about 270 mm, so when calculating for possible collisions, the robot was modeled as a circle of this radius. The robot only reaches these dimensions when measured from the centre to the rear of the robot. The radius of the front and the side of the robot was found to be closer to 220mm

CHAPTER 3

DESIGN

3.1 BASIC CLASSES

There are two basic classes at the heart of the pathfinding algorithm. The first and most commonly used is designated as the Node class. Primarily it is used to attach an x and y value together to define a location in 2 dimensional space. It is also implemented with several overloaded operators so that multiple nodes can be sorted by the algorithms built in to the C++ STL (Standard Template Library). The second basic class is the Node_link class. This class contains members defining two locations in 2 dimensional space, as well as the distance between them. This class is used to keep track of connections between two nodes. The complete source code for these classes can be found in Appendix A.

3.2 OBSTACLETRACKER

The ObstacleTracker class is used to keep track of any groups of Nodes. The class's primary purpose is to record the locations of any obstacles that are detected by the robot. In order to make searching for nodes near a particular coordinate faster, this class takes advantage of two classes in the STL, the map class and the vector class. The nodes are stored in a map of vectors of nodes. Which vector a given node is stored in depends on the location of that node. At the instantiation of an ObstacleTracker, the value *reg_size* must be provided which indicates the size of each geographical region that each vector covers. For instance, if *reg_size* is 120, then each vector in the map would only contain nodes that belong in a particular 120 by 120 square region. The units of this number are the same as whatever units the robot uses to report distance in. In the case of the Pioneer 2 robot, which this algorithm was tested on, it would be mm. New entries to the map are only added as they are needed. If a node is added to an instance of ObstacleTracker that is the first node in that region,

only at that time is an entry in the map for that region added. This means that the map in an unused instance of ObstacleTracker is is empty.

The primary way of adding new nodes to the map is through the *add_Ob(Node new_node)* function. This function will only add a new node to the map if the new node is greater than a distance of *inRange* away from all other nodes in the given instance of ObstacleTracker. The variable *inRange* is a value that is set preferably before the instance is used, although this is not necessary. The class ObstacleTracker can be set to promiscuous mode by setting a member flag. This flag completely disables the check for *inRange*, so no new nodes are rejected due to their proximity to another existing node.

The function *NodesNear(Node tocheck, double range)* returns a vector containing all the nodes from the regions that are within *range* of *tocheck*. If *inRange* is much smaller than *range*, then there exists the possibility entire regions that are not even partly within the defined distance are returned, which would be a source of inefficiency. However, *inRange* and *range* are typically on the same order, so this problem is only known to arise at one point during the algorithm. This will be discussed more fully with the pathfinding algorithm.

The ObstacleTracker class also has a function that is used to log all the points located in the map to a text file. This is used for debugging purposes so that the map of nodes can be visualized by Matlab. The complete code for ObstacleTracker can be found in Appendix B.

3.3 NODE_HANDLER

Node_handler is the class that performs most of the complex calculations in this algorithm and is responsible for generating the route to the goal. Only a single instance of it is created for the pathfinding algorithm. Among its members, it has one ObstacleTracker instance as well as two pointers to ObstacleTrackers. These two pointers point to ObstacleTrackers that are instantiated in the main body of the algorithm and they are used to keep track of the obstacles that are detected in the territory that the robot travels through. They monitor the robot's sensors and keep track of each new sonar echo that is returned, however one resets whenever a route is successfully generated. The ObstacleTracker that is internal to Node_handler keeps track of the position of the nodes that are generated during the pathfinding algorithm.

When in operation, Node_handler generates a list of random nodes that spans an area that contains the destination, the current robot location, as well as all of the detected obstacles. Once the nodes are generated, the algorithm checks over all the nodes and creates a Node_link between each of

the nodes that are within the distance *connec_within* of each other. The variable *connec_within* is a member of Node_handler. The Node_links created are stored in another member variable called *link_list*, which is a vector of Node_links. The value *connec_within* itself is calculated based on the area that the random node field covers, as well as the number of nodes that are generated. When the area decreases or the number of nodes increases, *connec_within* decreases. The opposite is also true. This keeps the average connectivity of the nodes about the same for any given area and number of nodes. The number is calculated first by figuring out the dimensions of a hexagon that would be required to tile the area of the node field if one were to use the same number of tiles as were nodes generated. The distance *connec_within* becomes the length of one side of one of the hexagons, multiplied by 3. The resulting links produced between nodes are sufficient for the purposes of this algorithm. The average connectivity, if there are no obstacles, is 5.3 links per node even with varied map sizes.

Once the links have been generated, they are checked against the known obstacles using the check_intercept() function. Any links that come within *R_radius* of a known obstacle are removed from *link_list*. This is because if the robot attempted to traverse those links, it would likely collide with an obstacle. *R_radius* is set to the effective radius of the robot, which was found to be 270mm, as stated earlier.

Figure 1 shows a node field with links and obstacles. The black circles indicate the collision regions surrounding detected obstacles and the light grey lines indicate links that pass through these areas. The dark lines indicate links that are not obscured and are suitable for travel.

Once link_list has been populated and checked for obstructions by check_intercept(), it is turned into a graph and run through an implementation of Dijkstra's algorithm. The implementation of Dijkstra's was not written expressly for this project and was produced by Shane Saunders of the University of Canterbury[1]. A Fibonacci heap was used to implement the priority queue of the shortest path algorithm, which gave relatively fast amortized running time of $O(E + V\log V)$, where E is the number of edges, and V is the number of vertices [5-7]. Typically, when the project was run, $V \approx 2000$ and $E \approx 10000$. Theoretically speaking, this is not the fastest that Dijkstra's algorithm could go, but the faster heaps required conditions that this project could not guarantee[1]. There are algorithms that are faster than Dijkstra even used with an ideal heap, but time required to run Dijkstra's was considered acceptable, and no other algorithms were implemented[8-23].

Once Dijkstra's algorithm is complete, Node_handler outputs a list of nodes that represent the nodes that should be traveled through to reach the target location.

The complete code to the Node_handler class can be found in Appendix C.

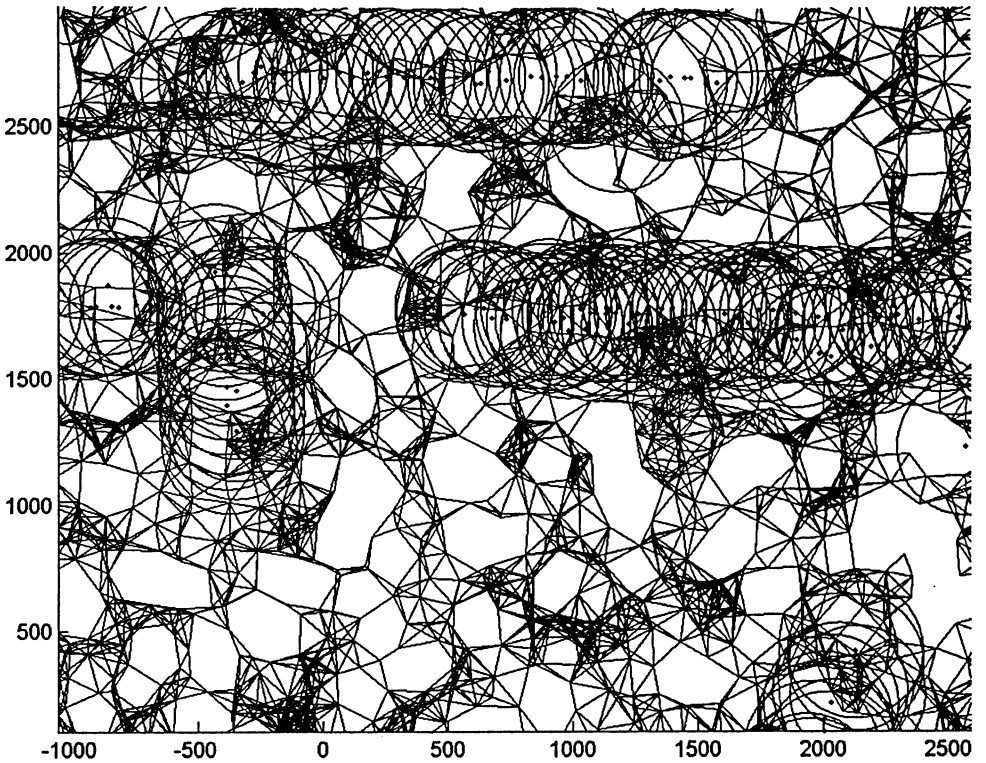

Figure 1 – Graphs with superimposed Obstacles

3.4 PATHFINDING ALGORITHM

The main algorithm is a loop that runs until either the robot has been shutoff, or the robot arrives at its destination. The contents of the loop can be divided into three sections. The first section is where the program queries each of the sonar sensors on the robot to find if they have new sonar information. If they do, the coordinates of the sonar echo are entered into two ObstacleTrackers, near_Tracker and ob_Tracker using Add_ob().

Once all the sensors have been queried, the robot updates its current location. This value is required for the next part of the algorithm.

The robot then checks if there are any obstructions between the current location and the next waypoint. If there are any obstructions, then the robot sets the *route_blocked* flag, sends the stop command to the robot, and will enter the route finding part of the loop. It is whenever the

route_blocked flag is set that the robot will actually begin reacting to obstacles detected around it[24-30].

Due to the design of the robot, it is possible for the robot to become stuck traveling in a circle and unable to a make a turn tight enough to arrive at the required waypoint. In this paper this is being called a circle stall. By comparing new current location with its previous known location, it figures out how far it has traveled, assuming a straight path. The distance measurement acts as a kind of timeout. If the robot travels too far without arriving at a waypoint, it assumes that it is stuck in a circle and resets the pathfinding algorithm to start fresh. The next waypoint it attempts to travel to will be in a different location.

Section 2 of the loop only happens for two cases. Either an obstruction was detected along the current path, or else this is the first iteration of the loop, so there is no current route to the destination. The algorithm first checks to see if it can find the route using the short cycle mode. The short cycle mode makes use of the existing node field and *link_list* without recreating them. This is done because the slowest operations in the algorithm are the generation of the links and checking them for obstructions. It speeds the algorithm if new node fields are only generated when a route cannot be found with the current field. The ObstacleTracker near_Tracker keeps track of the obstacles that were discovered in the time from when the last route was found and up until the present. These are the obstacles that are used when check_intercept() is run on the already existing *link_list*. Dijkstra's algorithm is run on the graph defined by link_list, and if a route cannot be found, then the long cycle mode begins.

The long cycle mode begins with the erasure of the existing node field, and a new one being generated. Prior to the actual generation of the nodes, the dimensions of the node field are updated so that it will include all the area that contains any known obstacles, as well as the current location of the robot and the destination. The links are then generated for this node field and Dijkstra's is applied to the graph. If a route cannot be found, then the density of the node field is increased, and another attempt is made to locate the route. The algorithm iterates through this loop until a route is found. Once the route is found, the third section of the algorithm begins.

The third and final section of the algorithm reads the list of waypoints from the route that the robot must travel through to get to the destination and checks to see if it is currently at the next waypoint on the list. If it is, then the robot's new local goal is set to the next waypoint on the list. Since the distance between waypoints is usually on the same order of R_radius, the course that the robot would take if it followed the waypoints exactly would be extremely chaotic, requiring frequent and rapid course corrections. For non-ideal localization, this causes a significant increase in error from odometric drift, as well as increasing the chance of circle stalls when using any kind of

localization. In order to limit this effect, before the robot commits to a heading towards a particular waypoint, it first checks all the waypoints sequentially along the route and finds whether or not it can reach them without running into any known obstacles. It chooses the furthest reachable waypoint, and sets that as its next local goal. This results in a series of straight lines that approximates the more random initial route. Because the distance being checked becomes relatively far compared to *R_radius*, the function NodesNear() will return nodes that are too far from the link to possibly get in the way. This is not efficient, but it is acceptable because this operation does not occur often enough to appreciably slow the algorithm down. Once the furthest waypoint is chosen, the initial stop command from section one is cleared, the robot starts moving again, and the algorithm starts the loop again. The full code for this algorithm can be found in Appendix D.

CHAPTER 4

RESULTS

The robot was tested in its ability to navigate in two different kinds of geographies. First it was tested in its ability to navigate through an open space with scattered square obstacles(See Fig. 2). Second it was tested in its ability to navigate through simulated series of empty rooms and corridors(See Fig. 3). In both cases, the robot would start very slowly. First, it would have to develop the initial version of *link_list* from scratch, finding the route in long cycle mode. Once the route was created, the robot would attempt to traverse it until it encountered an obstacle along its current path. At first, this would happen several times in quick succession before the nearby obstacles were more fully mapped out. The robot would detect an obstacle and be forced in to short cycle pathfinding. Once the new route was found, the robot would begin moving again, only to almost immediately discover another obstacle that blocked its path. Once the nearby obstacles had been fully mapped out, the robot would move more smoothly towards its target. The use of the short cycle lets this phase of the pathfinding happen fairly quickly. The smoother, faster movement would continue, usually until the robot turned a corner and made sonar contact with a new obstacle, or a previously unseen side of a known obstacle. At that point, the long cycle would begin again, or possibly another series of short cycle loops.

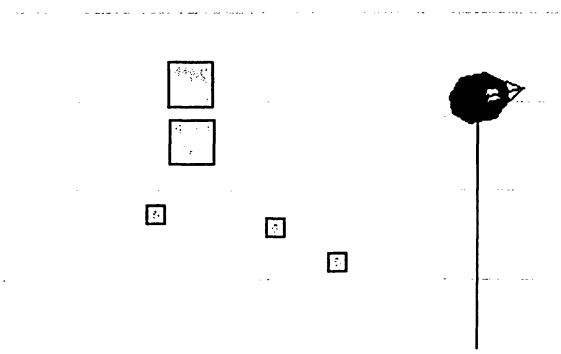

Figure 2 - Simulation map of gap

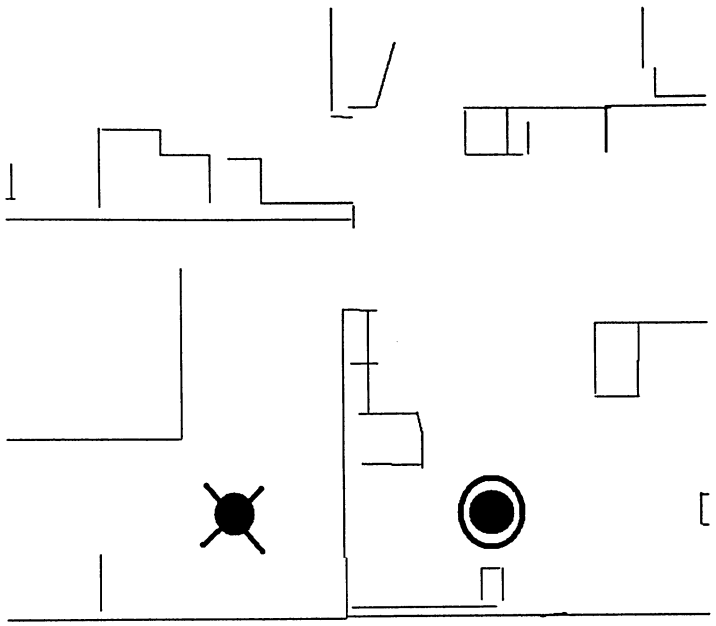

Figure 3 - Simulation Map of Doorway

In a particular experiment, the robot traveled through the arrangement of blocks shown in figure 2. The robot traveled from the left side of the blocks to the right, passing between a large block and a small block. Figure 4 shows the data the robot logged as it traveled. The series of black dots represents the actual path the robot took, while the straight black lines show the desired path. The circles represent the regions where collisions with obstacles will occur, the same as in figure 1.

An example of the results from the second geography are shown in figure 5. In it, the robot successfully exits from the room on the left, proceeds through a doorway, and enters a second room. In figure 3 the actual simulation map, the robot is traveling from the crossed dot to the circled dot.

The algorithm showed better ability at navigating through the field of scattered obstacles than at navigating through a corridor. Barring problems that derived from inaccurate kinematic modeling, the algorithm was quite reliable.

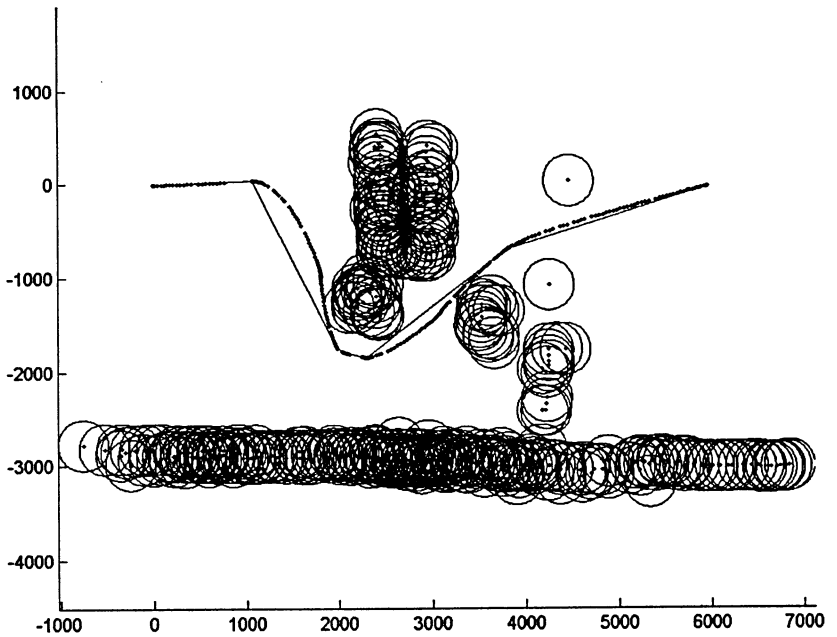

Figure 4 – Robot traveling through gap

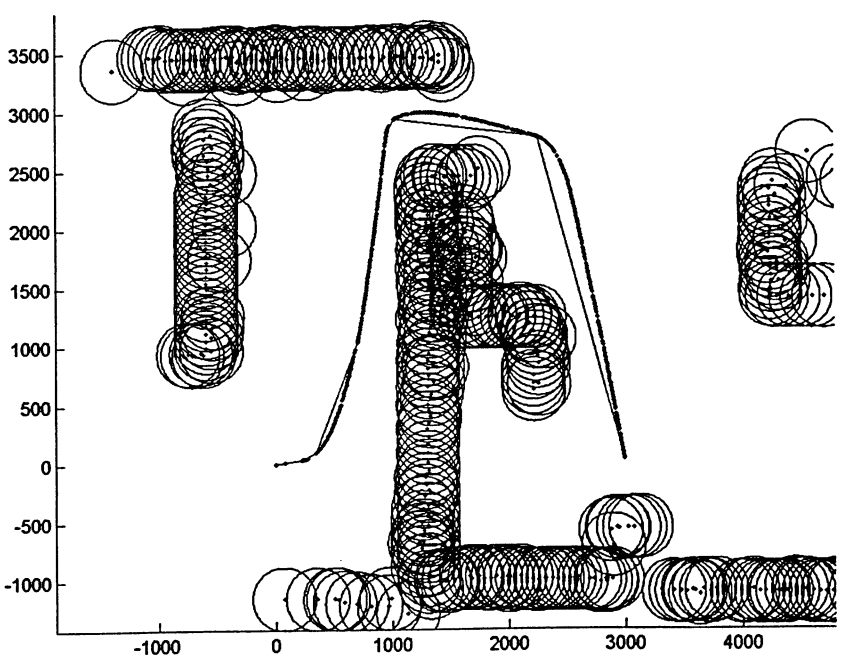

Figure 5 – Robot traveling through doorway

CHAPTER 5

FURTHER RESEARCH

Because of the real world dimensions of the robot are not perfectly circular, it is possible for the robot to be within *R_radius* of an obstacle and not actually collide with it. For example, this can occur when the robot makes a turn around the corner of a rectangular obstacle and drives to within R_radius of the previously unseen side before it has properly registered it in an ObstacleTracker. The robot will stop upon detecting the obstacle, but subsequent loops of the pathfinding algorithm will never find a new route to the destination. This is because any links that connect to the node that represents the robots location will be removed when check_intercept() runs. Making a special case for this node, and ignoring its proximity to an obstacle is not a valid solution because then links that could cause collisions would not be culled by check_intercept(). If this was done, then the robot might be able to proceed, or it could simply collide with a wall. A mechanism that could deal with this situation would increase the reliability of the robot.

In figure 4 it can be seen that the robot's actual path does not follow the exact course that is chosen by the algorithm. This is due to the use of a simplistic model of the robot's kinematics. The algorithm would benefit from the integration of the ego-kinematic space representation developed by Minguez *et al*[31-33], or any other appropriate kinematic model[34-35]. A simpler solution might be to change the movement commands given to the robot so that instead of performing translation and rotation maneuvers at the same time, it turns to face the next waypoint, then drives in a straight line, in two separate steps, giving the robot a kind of pseudo holonomic ability. This cannot be done currently due to problems with the ARIA libraries, which require location and orientation information be derived indirectly.

The robots inability to stop instantaneously is also a problem. It is possible for the robot to roll into an obstacle, even after the stop command is sent, due to its deceleration limitations. A proper kinematic model would help to reduce this effect, but would require greater predictive ability on the part of the algorithm in calculating when and where the robot could collide with an obstacle. In figure 4 at approximately (1000,0), a gap where the position of the robot is not logged can be seen.

The logged position on the left side of the gap is where the robot was when the stop command was sent and the logged position on the right side of the gap is where the robot rolled to before coming to a complete stop. The robot stayed at that position until a new route to the destination was generated. Once it was underway again, it resumed logging its positions.

CHAPTER 6

CONCLUSIONS

This algorithm is successful at route finding, with some problems that still need to be resolved. The primary difficulty with this algorithm is in avoiding obstacles that the robot approaches at an oblique angle. This algorithm has a strength in its ability to take a global view to any given navigation decision. This means that there is no way misleading information at a localized level can cause incorrect decisions at the global level[36-42]. This geographical awareness is an excellent advantage.

Generating the links also incurs a time penalty that prevents the robot from arriving at its destination quickly, however this is a problem that can probably be remedied through improved hashing techniques.

In closing, this is an algorithm that shows promise but definitely requires further development.

APPENDIX A

Node_structs.h Source code

```
#pragma once

#include "stdio.h"


using namespace::std;

//Node class is used to store points in 2D space
class Node {
public:
    double x;
    double y;
      double dist_fromcentre;
      double dist_fromnode;
      //This value keeps track of the node's position in it's particular
vector when it is placed
      //in a map.
      int index;

      Node(void);
      bool operator==(Node myNode);
      bool operator<(const Node myNode) const;
      string toString(void);

};
Node::Node(void)
{
      x=0;
      y=0;
      index=0;
}
bool Node::operator==(Node myNode)
{
      if((this->x==myNode.x)&&(this->y==myNode.y))
            return true;
      else
            return true;
}
//A node further form point (0,0) is greater than one that is closer
bool Node::operator<(const Node myNode) const
{
```

```
	double dist1;
	double dist2;
	dist1=pow(this->x,2)+pow(this->y,2);
	dist2=pow(myNode.x,2)+pow(myNode.y,2);
	return dist1<dist2;
}

//Returns a string of the paired x and y values
string Node::toString(void)
{
	char outTextX[20];
	char outTextY[20];
	string  outString;

	sprintf(outTextX,"%f",x);
	sprintf(outTextY,"%f",y);

	outString=string(outTextX) + "\t" + string(outTextY);
	return outString;
}

//Node_link class stores a connection between two points in 2D space
struct Node_link {
	double x1;
	double y1;
	double x2;
	double y2;
	double dist;
	Node_link(void);
	Node_link(double new_x1, double new_y1, double new_x2, double
new_y2);
};
Node_link::Node_link(void)
{
	x1=0;
	x2=0;
	y1=0;
	y2=0;
}
Node_link::Node_link(double new_x1, double new_y1, double new_x2, double
new_y2)
{
	x1=new_x1;
	y1=new_y1;
	x2=new_x2;
	y2=new_y2;
}
//function used to sort Nodes
bool comp_node (Node i,Node j) { return
(i.dist_fromcentre<j.dist_fromcentre); }
bool comp_nodefromnode (Node i, Node j) { return
(i.dist_fromnode<j.dist_fromnode);}

//Class for ObstacleTracker to use to file
class Region{
public:
	double i;
```

```
	double j;
	bool operator==(Region myRegion);
	bool operator<(const Region myRegion) const;
};
bool Region::operator==(Region myRegion)
{
	if((this->i==myRegion.i)&&(this->j==myRegion.j))
		return true;
	else
		return false;
}
//< required for sorting operations
bool Region::operator<(const Region myRegion) const
{

	if(i<myRegion.i)
		return true;
	else if(i>myRegion.i)
		return false;
	else //if(i==myRegion.i)
	{
		if(j<myRegion.j)
			return true;
		else if(j>myRegion.j)
			return false;
		else
			return false;
	}

}

//DistanceTracker keeps track of how far the robot travels
class DistanceTracker {
public:
	DistanceTracker(void);
	DistanceTracker(double startwith);
	void new_waypoint(double x, double y);
	void resetLeg(void);
	double getLeg(void);
	double getTotal(void);
	void clearAndReset(void);

private:
	double total_traveled;
	double leg_traveled;
	bool start;
	double lastpointx;
	double lastpointy;

};
DistanceTracker::DistanceTracker(void)
{
	total_traveled=0;
	leg_traveled=0;
	lastpointx=0;
	lastpointy=0;
	start=true;
```

```
}
DistanceTracker::DistanceTracker(double startwith)
{
	total_traveled=startwith;
	leg_traveled=0;
	lastpointx=0;
	lastpointy=0;
	start=true;
}
//Adds distance to (x,y) to running total
void DistanceTracker::new_waypoint(double x, double y)
{
	if(start)
	{
		lastpointx=x;
		lastpointy=y;
	}
	else
	{
		double dist=sqrt(pow(lastpointx-x,2)+pow(lastpointy-y,2));
		lastpointx=x;
		lastpointy=y;
		total_traveled=total_traveled+dist;
		leg_traveled=leg_traveled+dist;
	}
	start=false;
}
void DistanceTracker::resetLeg(void)
{
	leg_traveled=0;
}
double DistanceTracker::getLeg(void)
{
	return leg_traveled;
}
double DistanceTracker::getTotal(void)
{
	return total_traveled;
}
void DistanceTracker::clearAndReset(void)
{
	total_traveled=0;
	leg_traveled=0;
	lastpointx=0;
	lastpointy=0;
	start=true;
}
```

APPENDIX B

ObstacleTracker class source code

```
#pragma once
#include <sstream>
#include <fstream>
#include <iostream>
#include "time.h"
#include "math.h"
#include <algorithm>
#include <vector>
#include "Node_structs.h"
using namespace::std;

class ObstacleTracker
{
public:
	//Vector of obstacles
	vector<Node> Ob_list;
	//Internal buffer distance
	//If new obstacle is within this distance, to another node, then
	//it is not added to list
	double inRange;
	double outRange;
	//Size of regions that the obstacles are assigned to
	double reg_size;
	//if NoQ=true, then Qualify routine has no effect
	//all incoming obstacles are added to Ob_Map;
	bool NoQ;

	double max_x, min_x, max_y, min_y;
	bool first_ob;

	//Map contains list of obstacles
	//they are sorted according to region in order to speed the
	//location process
	map<Region,vector<Node>> Ob_Map;

	ObstacleTracker(void);
	~ObstacleTracker(void);
	void Add_ob(double x, double y);
	//Version of Add_ob returns the index location of the added node.
	//Returns -1 if the node is not added to the map
	//int
```

```
	void Add_ob(Node new_node);
	void New_MaxMin(Node to_check);
	void setinRange(double new_inRange);
	void setoutRange(double new_outRange);
	void set_reg_size(double new_reg_size);
	void setNoQ(bool newNoQ);
	bool Qualify(double x, double y);
	bool Qualify(double x ,double y, vector<Node> * near_vect);
	void listToFile(char * obfilename);
	void readFromFile(char * obfilename);
	void Classify(void);
	Region AssignReg(double x, double y);
	vector<Node> NodesNear(Node tocheck,double range);
	vector<Node> AllNodes();
	void clear();
	void PrintOb_Map(void);

};
ObstacleTracker::ObstacleTracker(void)
{
	Ob_list.clear();
	inRange=50;
	outRange=1000;
	reg_size=700;
	NoQ=false;
	max_x=0;
	min_x=0;
	max_y=0;
	min_y=0;
	first_ob=true;
}

ObstacleTracker::~ObstacleTracker(void)
{
}

void ObstacleTracker::Add_ob(double x, double y)
{
	Node temp_node;
	//Initializing vector of obstcles near the new obstacle
	vector<Node> near_vect;
	vector<Node> temp_vect;
	Region temp_reg;

	temp_node.x=x;
	temp_node.y=y;

	if(!NoQ)
		near_vect=NodesNear(temp_node,inRange);
	if(NoQ || Qualify(x,y,&near_vect))
	{
		//Getting region for new obstacle to be assigned to
		temp_reg=AssignReg(x,y);
		if(Ob_Map.find(temp_reg)==Ob_Map.end())
			Ob_Map[temp_reg]=temp_vect;
		New_MaxMin(temp_node);
		Ob_Map[temp_reg].push_back(temp_node);
```

```
            //New_MaxMin(temp_node);
        }
}

void ObstacleTracker::Add_ob(Node new_node)
{
        //Initializing vector of obstcles near the new obstacle
        vector<Node> near_vect;
        vector<Node> temp_vect;
        Region temp_reg;
        int index=-1;

        if(!NoQ)
                near_vect=NodesNear(new_node,inRange);
        if(NoQ || Qualify(new_node.x,new_node.y,&near_vect))
        {
                //Getting region for new obstacle to be assigned to
                temp_reg=AssignReg(new_node.x,new_node.y);
                if(Ob_Map.find(temp_reg)==Ob_Map.end())
                        Ob_Map[temp_reg]=temp_vect;
                //Places index value in
                index=Ob_Map[temp_reg].size();
                //index=Ob_Map[temp_reg].size();
                Ob_Map[temp_reg].push_back(new_node);
                New_MaxMin(new_node);
        }
        return; //index;
}
void ObstacleTracker::New_MaxMin(Node to_check)
{
        if(first_ob)
        {
                max_x=to_check.x;
                min_x=to_check.x;
                max_y=to_check.y;
                min_y=to_check.y;
                first_ob=false;
        }
        else
        {
                //cout<<to_check.x<<"\t"<<to_check.y<<endl;
                if(to_check.x>max_x)
                        max_x=to_check.x;
                else if(to_check.x<min_x)
                        min_x=to_check.x;
                if(to_check.y>max_y)
                        max_y=to_check.y;
                else if(to_check.y<min_y)
                        min_y=to_check.y;
        }
}

void ObstacleTracker::setinRange(double new_inRange)
{
        inRange=new_inRange;
}
void ObstacleTracker::setoutRange(double new_outRange)
```

```
{
	outRange=new_outRange;
}
void ObstacleTracker::set_reg_size(double new_reg_size)
{
	reg_size=new_reg_size;
}
void ObstacleTracker::setNoQ(bool newNoQ)
{
	NoQ=newNoQ;
}
//Checks if the obstacle represented by x and y is within inRange of any
existing obstacles
bool ObstacleTracker::Qualify(double x, double y)
{
	vector<Node>::iterator it;
	vector<Node>::reverse_iterator rit;
	double dist=0;

	it=Ob_list.begin();
	//checks in reverse order so that most recent entries are checked
first.
	//This loses its effectiveness if the robot is backtracking through
previously scanned areas
	for(rit=Ob_list.rbegin(); rit!=Ob_list.rend();++rit)
	{
		dist=sqrt(pow((rit->x)-x,2)+pow((rit->y)-y,2));
		if(dist<inRange)
			return false;
	}
	return true;
}

bool ObstacleTracker::Qualify(double x, double y, vector<Node> *
near_vect)
{
	//vector<Node>::iterator it;
	vector<Node>::reverse_iterator rit;
	double dist=0;

	//it=Ob_list.begin();
	//checks in reverse order so that most recent entries are checked
first.
	//This loses its effectiveness if the robot is backtracking through
previously scanned areas
	//for(rit=Ob_list.rbegin(); rit!=Ob_list.rend();++rit)
	for(rit=near_vect->rbegin();rit!=near_vect->rend();rit++)
	{
		dist=0;
		dist=sqrt(pow((rit->x)-x,2)+pow((rit->y)-y,2));
		if(dist<inRange)
			return false;
	}
	return true;
}
void ObstacleTracker::listToFile(char * obfilename)
{
```

```
	//Opening log files
	fstream ObMapfile(obfilename,ios::out);

	map<Region,vector<Node>>::iterator m_it;
	vector<Node>::iterator ob_it;

	for(m_it=Ob_Map.begin();m_it!=Ob_Map.end();m_it++)
	{
		//cout<<"Region: "<<m_it->first.i<<"\t"<<m_it->first.j<<endl;
		for(ob_it=m_it->second.begin();ob_it!=m_it-
>second.end();ob_it++)
		{
			ObMapfile<<ob_it->x<<"\t"<<ob_it->y<<endl;

		}
	}
	//Closing log file
	ObMapfile.close();
}
//Reads in a list of obstacles from file obfilename.  These obstacles are
not assigned to the
//container map, so they cannot be seen by the Node_handler's
check_intercept() method.
void ObstacleTracker::readFromFile(char * obfilename)
{
	fstream ObMapfile(obfilename,ios::in);
	vector<Node>::iterator it;

	//fstream obfileIn(inObstFile,ios::in);
	Node new_obstac;
	double xx;
	double yy;
	//Try x and y as string, then use double atof(string) to convert.
	cout<<"Reading file: "<<obfilename<<endl;
	//while(obfileIn>>xx)
	while(!ObMapfile.eof())
	{
		//bool d=obfileIn.setf(std::ios::eof());
		//bool d=cin.setf(ios::eof);
		ObMapfile>>xx;
		//cout<<"xx: "<<xx<<endl;

		//new_obstac.x=atof(xx.c_str());
		new_obstac.x=xx;
		ObMapfile>>yy;
		//new_obstac.y=atof(yy.c_str());
		new_obstac.y=yy;
		Ob_list.push_back(new_obstac);
	}
	ObMapfile.close();
}
//Applies the Qualify check to all the nodes read in by readFromFile and
puts them in
//Ob_Map
void ObstacleTracker::Classify(void)
{
	vector<Node>::iterator it;
```

```
	//vector<Node> temp_vect;
	//Region temp_reg;
	for(it=Ob_list.begin();it!=Ob_list.end();it++)
	{
		Add_ob(it->x,it->y);
		//temp_reg=AssignReg(it->x,it->y);
		//if(Ob_Map.find(temp_reg)==Ob_Map.end())
		//    Ob_Map[temp_reg]=temp_vect;
		//Ob_Map[temp_reg].push_back(*it);
	}
	//cout<<"End classify"<<endl;
	//cout<<max_x<<"\t"<<min_x<<"\t"<<max_y<<"\t"<<min_y<<endl;
}
//Returns the region that contains the input arguments
Region ObstacleTracker::AssignReg(double x, double y)
{
	Region myReg;
	myReg.i=floor(x/reg_size);
	myReg.j=floor(y/reg_size);
	return myReg;
}
//Returns a vector containing all the nodes within "range" of "tocheck"
vector<Node> ObstacleTracker::NodesNear(Node tocheck,double range)
{
	vector<Node> NearNodes;
	Region checknear=AssignReg(tocheck.x,tocheck.y);
	Region up, down, left, right;
	double uspan, dspan, lspan, rspan;
	Region temp_region;
	vector<Node>::iterator it;
	if(2*range<reg_size)
	{
		up=AssignReg(tocheck.x,tocheck.y+range);
		down=AssignReg(tocheck.x,tocheck.y-range);
		left=AssignReg(tocheck.x+range,tocheck.y);
		right=AssignReg(tocheck.x-range,tocheck.y);
		//If all the outermost regions are the same, then only the
obstacles in the same region
		//have to be returned, therefore span=0;
		if((up==down)&&(up==left)&&(up==right))
		{
			uspan=0;
			dspan=0;
			lspan=0;
			rspan=0;
		}
		else
		{
			uspan=max(checknear.j,up.j)-checknear.j;
			dspan=checknear.j-min(checknear.j,down.j);

			rspan=max(checknear.i,right.i)-checknear.i;
			lspan=checknear.i-min(checknear.i,left.i);
		}
	}
	else
	{
```

```
            int span=(int)ceil(range/reg_size);
            uspan=span;
            dspan=span;
            lspan=span;
            rspan=span;
      }


      //cout<<"span: "<<span<<endl;
      for(int i=checknear.i-lspan;i<=checknear.i+rspan;i++)
      {
            for(int j=checknear.j-dspan;j<=checknear.j+uspan;j++)
            {
                  //temp_region=AssignReg(i,j);
                  temp_region.i=i;
                  temp_region.j=j;
                  if(Ob_Map.find(temp_region)!=Ob_Map.end())
                  {

      for(it=Ob_Map[temp_region].begin();it!=Ob_Map[temp_region].end();it+
+)
                        {
                              NearNodes.push_back(*it);
                        }

                  }
            }
      }
      if(NearNodes.size()>0)
      {
            //cout<<"size that should be returned:
"<<NearNodes.size()<<endl;
      }
      return NearNodes;
}
//Returns a vector containing all the nodes that ObstacleTracker is
recording
vector<Node> ObstacleTracker::AllNodes()
{
      vector<Node> temp_vect;
      map<Region,vector<Node>>::iterator m_it;
      vector<Node>::iterator it;

      for(m_it=Ob_Map.begin();m_it!=Ob_Map.end();m_it++)
      {
            for(it=m_it->second.begin();it!=m_it->second.end();it++)
            {
                  temp_vect.push_back(*it);
            }
      }

      return temp_vect;
}
//Erases contents of ObstacleTracker
void ObstacleTracker::clear(void)
{
      Ob_Map.clear();
```

```
	max_x=0;
	min_x=0;
	max_y=0;
	min_y=0;
	first_ob=true;
}
//Prints Ob_Map to console for debugging purposes
void ObstacleTracker::PrintOb_Map(void)
{
	map<Region,vector<Node>>::iterator m_it;
	vector<Node>::iterator ob_it;
	for(m_it=Ob_Map.begin();m_it!=Ob_Map.end();m_it++)
	{
		cout<<"Region: "<<m_it->first.i<<"\t"<<m_it->first.j<<endl;
		for(ob_it=m_it->second.begin();ob_it!=m_it-
>second.end();ob_it++)
		{
			cout<<"Obstacle: "<<ob_it->x<<"\t"<<ob_it->y<<endl;

		}
	}
}
```

APPENDIX C

Node_handler class source code

```
#pragma once
#include <sstream>
#include <fstream>
#include <iostream>
#include "time.h"
#include "math.h"
#include <algorithm>
#include <vector>
#include <map>
#include "Node_structs.h"
#include "ObstacleTracker.h"

#include "dijkstra\\dgraph.h"
#include "dijkstra\\dijkstra.h"
#include "dijkstra\\fheap.h"
#include "dijkstra\\heap.h"

using namespace::std;

#define hexArea 2/(3*sqrt(3.0))

class Node_handler
{
public:
	//Variables
	int num_nodes;
	int total_nodes;
	int map_size_x, map_size_y;
	double connec_within;
	//Effective radius of robot.  Used to calculate collision distances.
	double R_radius;


	//Vector of links between nodes
	vector <Node_link> link_list;
	//Vector of links that are clear of obstacles
	//vector <Node_link> clear_links;

	vector<Node> legacy_nodes;
	vector<Node_link> legacy_links;
```

```
	//Vector of Nodes
	vector <Node> node_list_vect;

	//Vector of obstacle points
	//Will be removed when the ObstacleTracker class is integrated
	vector <Node> ob_vect;

	//Current robot location
	Node curr_robloc;

	//Where the robot is trying to get to.
	//The dijkstra's algorithm finds the route between this and
curr_robloc
	Node destination;

	//Center of randomly generated nodes;
	Node centre;

	//Pointer to related obstacle tracker
	ObstacleTracker * in_Tracker;

	//Hashes the randomly generated nodes by location.  It allows the
proximity search to
	//run faster
	ObstacleTracker node_Tracker;

	//Keeps track of new opstacles.  Specifically, it stores all the
detected obstacles
	//Since the most recently generated route to the destination was
created
	ObstacleTracker * nearOb_Tracker;

	//Map for keeping track of node key values
	map<Node, int> node_key;

	stack <int> outroute;


	//methods
	Node_handler(void);
	Node_handler(Node new_robloc);
	~Node_handler(void);

	void set_centre(Node new_robloc);
	void set_centre(double x, double y);
	void set_robloc(double x, double y);
	void set_destination(Node new_destination);
	void set_destination(double x, double y);
	void set_numnodes(int new_num);
	void set_map_size(double new_map_size_x, double new_map_size_y);
	void set_radii(double new_radii);
	void auto_connecDist(void);
	void auto_config(bool use_dest=true);
	int big_rand(int rep);
	//void genNodes(void);
	void genNodes_int(bool overwrite);
	//void genLinkProx(void);
```

```
	void genLinkProx2(void);
	void genLinkProx_NewNode(double x, double y);
	void testout(void);
	void print_node_vector(vector<Node> vector_out);
	void print_node_vector(char * nodefile);
	void print_link_list(vector<Node_link> links_out);
	void print_link_list(char * linkfile);
	void calc_dist_fromNode(vector<Node> * Nodelist, Node myNode);
	void Add_Link(vector <Node_link> * vlist, Node origin, Node dest);
	void add_Legacy(double x1, double y1, double x2, double y2);
	void integrate_LNodes(void);
	void integrate_LLinks(void);
	void getObstacles(char * inObstFile);
	void getTracker(ObstacleTracker * new_tracker);
	void getTrackerNear(ObstacleTracker * new_tracker);
	bool find_shortroute(void);
	void print_shortroute(char * shortmapfile);
	vector<Node> out_shortroute(void);

	void print_obstacles(void);
	void print_obstacles(char * obfile);
	void print_legacy(char *lfile);

	void check_intercept(void);
	void check_intercept(vector<Node_link> * short_list);
	void check_intercept_newObs(void);
	bool check_intercept_single(Node_link * check_link, Node * obstac,
double range);
	bool route_blocked(double x1, double y1, double x2, double y2);

	//void check_intercept2(void);
	double dist_between(double x1, double y1, double x2, double y2);
	double dist_between(Node mynode1, Node mynode2);
	//bool operator==(Node_handler * second);


};
//bool Node_handler::operator==(Node_handler * second)
//{
//    return true;
//}
Node_handler::Node_handler(void)
{
		//Initialize random number generator
	srand ( time(NULL) );
	//srand(5);

	num_nodes=100;
	total_nodes=0;
	map_size_x=30;
	map_size_y=30;
	connec_within=1.2;
	R_radius=1;
	curr_robloc.x=0;
	curr_robloc.y=0;
	node_Tracker.setinRange(0);
	node_Tracker.set_reg_size(270*2);
```

```
      //Makes it so any new nodes added to the node_Tracker are not
subject to
      //proximity exlcusion
      node_Tracker.setNoQ(true);


}
Node_handler::Node_handler(Node new_robloc)
{
            //Initialize random number generator
      srand ( time(NULL) );

      num_nodes=100;
      map_size_x=30;
      map_size_y=30;
      connec_within=1.2;
      R_radius=1;

      curr_robloc.x=new_robloc.x;
      curr_robloc.y=new_robloc.y;

}

Node_handler::~Node_handler(void)
{
      //delete dijkstra;
    //delete myDgraph;
      //delete [] d;
}
//Sets centre of next node field that is generated
void Node_handler::set_centre(Node new_robloc)
{
      set_centre(new_robloc.x,new_robloc.y);
}
//Sets centre of the next node field that is generated
void Node_handler::set_centre(double x, double y)
{
      centre.x=x;
      centre.y=y;
}
//Sets current location of robot
void Node_handler::set_robloc(double x, double y)
{
      curr_robloc.x=x;
      curr_robloc.y=y;
}
//Sets the location that is searched for
void Node_handler::set_destination(Node new_destination)
{
      set_destination(new_destination.x,new_destination.y);
}
//Sets the location that is searched for
void Node_handler::set_destination(double x, double y)
{
      destination.x=x;
      destination.y=y;
}
```

```
//Sets size of next node field to be generated
void Node_handler::set_numnodes(int new_num)
{
      num_nodes=new_num;
}
//Sets the size of the next node field to be generated.
void Node_handler::set_map_size(double new_map_size_x, double
new_map_size_y)
{
      map_size_x=new_map_size_x;
      map_size_y=new_map_size_y;
}
//Sets the distance used to determine collision distance.
//Should be set as the maximum radii of the robot being controlled.
//For Pioneer 2 robot, this value should be 270mm
void Node_handler::set_radii(double new_radii)
{
      R_radius=new_radii;
}
//Sets the maximum connection distance between the nodes automatically.
//Value is calcualated based on the theoretical average distance between
the
//randomly generated nodes.
void Node_handler::auto_connecDist(void)
{
      connec_within=1.5*2*sqrt(hexArea*(4*map_size_x*map_size_y)/num_nodes
);
      cout<<"connect: "<<connec_within<<endl;
      //cout<<hexArea<<endl;
      node_Tracker.set_reg_size(2.1*connec_within);
}
//Sets the size of the map based on the known obstacles, as well as the
location
//of the robot and the destination
void Node_handler::auto_config(bool use_dest)
{
      double max_x, min_x, max_y, min_y;
      max_x=in_Tracker->max_x;
      min_x=in_Tracker->min_x;
      max_y=in_Tracker->max_y;
      min_y=in_Tracker->min_y;
      cout<<max_x<<"\t"<<min_x<<"\t"<<max_y<<"\t"<<min_y<<endl;

      if(curr_robloc.x>max_x)
            max_x=curr_robloc.x;
      if(curr_robloc.x<min_x)
            min_x=curr_robloc.x;
      if(curr_robloc.y>max_y)
            max_y=curr_robloc.y;
      if(curr_robloc.y<min_y)
            min_y=curr_robloc.y;

      if(use_dest)
      {
            if(destination.x>max_x)
                  max_x=destination.x;
```

```
            if(destination.x<min_x)
                  min_x=destination.x;
            if(destination.y>max_y)
                  max_y=destination.y;
            if(destination.y<min_y)
                  min_y=destination.y;
      }

      //Setting mapsize and centre variables to contain all obstacles, as
well as the source
      //and destination
      cout<<max_x<<"\t"<<min_x<<"\t"<<max_y<<"\t"<<min_y<<endl;
      //map_size_x=1.1*((max_x-min_x)/2);
      //map_size_y=1.1*((max_y-min_y)/2);

      map_size_x=2*R_radius+((max_x-min_x)/2);
      map_size_y=2*R_radius+((max_y-min_y)/2);

      cout<<"map x: "<<map_size_x<<endl;
      cout<<"map y: "<<map_size_y<<endl;

      centre.x=(max_x+min_x)/2;
      centre.y=(max_y+min_y)/2;
      cout<<"centre.x: "<<centre.x<<endl;
      cout<<"centre.y: "<<centre.y<<endl;


}
//Produces a large random number
int Node_handler::big_rand(int rep)
{
      int bigR=0;
      for(int i=0;i<rep;i++)
      {
            bigR=bigR+rand();
      }
      return bigR;
}
//void Node_handler::genNodes(void)
//{
//    //#define RAND_MAX 0x7fff
//    //Initializing temp nodes
//    Node new_node;
//
//    //Generating list of nodes
//    for (int i=0;i<num_nodes;i++)
//    {
//          //cout<<"i: "<<i<<endl;
//          new_node.x=(double)(rand()%(map_size*2*100) -
map_size*100)/100 + curr_robloc.x;
//          new_node.y=(double)(rand()%(map_size*2*100) -
map_size*100)/100 + curr_robloc.y;
//          new_node.dist_fromcentre=dist_between(curr_robloc,new_node);
//          new_node.dist_fromnode=0;
//          node_list_vect.push_back(new_node);
```

```
//
      //cout<<node_list[i].x<<"\t"<<node_list[i].y<<"\t"<<node_list[i].dis
t_fromcentre<<endl;
//    }
//    //Adding current location and destination nodes
//    node_list_vect.push_back(curr_robloc);
//    node_list_vect.push_back(destination);
//}

void Node_handler::genNodes_int(bool overwrite)
{
      if(overwrite)
      {
            node_list_vect.clear();
            node_Tracker.Ob_Map.clear();
            link_list.clear();
            total_nodes=0;
      }
      //double av_x=0;
      //double av_y=0;
      Node temp_node;
      int temp_index=0;
      //cout<<"temp_node declared"<<endl;
      for(int i=0;i<num_nodes;i++)
      {
            //cout<<"i: "<<i<<endl;
            temp_node.x=(double)(big_rand(10)%(2*map_size_x)-
map_size_x+centre.x);
            //av_x=av_x+temp_node.x;
            //cout<<"x generated"<<endl;
            temp_node.y=(double)(big_rand(10)%(2*map_size_y)-
map_size_y+centre.y);
            //av_y=av_y+temp_node.y;
            //cout<<"y generated"<<endl;
            temp_node.dist_fromcentre=dist_between(curr_robloc,temp_node);
            temp_node.dist_fromnode=0;

            //Puts node in Map where it is organized by location
            //temp_index=
            node_Tracker.Add_ob(temp_node);
            //Puts node in vector of nodes
            //if(temp_index!=-1)
            //{
            //    temp_node.index=temp_index;
            //    node_list_vect.push_back(temp_node);
            //}

      }
      //cout<<"temp_x: "<<temp_node.x<<endl;
      //av_x=av_x/num_nodes;
      //av_y=av_y/num_nodes;
      //cout<<"Average of randomizer: (x,y): "<<av_x<<"\t"<<av_y<<endl;
      //Adding current location and destination nodes

      //node_list_vect.push_back(curr_robloc);
      //node_list_vect.push_back(destination);
      if(overwrite)
```

```
	{
		node_Tracker.Add_ob(curr_robloc);
		node_Tracker.Add_ob(destination);
		total_nodes=total_nodes+2;
	}

	//Keeping track of total number of nodes
	total_nodes=total_nodes+num_nodes;
}

/*
void Node_handler::genLinkProx(void)
{
	//Create vectors for node sorting
	vector<Node> node_list_trunc;
	vector<Node> node_list_temp;
	//node_list_temp.assign(node_list_vect.begin(),node_list_vect.end())
;
	node_list_temp=node_Tracker.AllNodes();
	node_list_vect=node_Tracker.AllNodes();

	//For iterating through node_list
	vector<Node>::iterator it;


	bool link_found=false;

	//Vector list of links between nodes
	//vector<Node_link> link_list;
	vector<int> connec_nodes;

	//Sorts node list by comparing distance from centre
	//node_list_vect doesn't change order after this
	sort(node_list_vect.begin(),node_list_vect.end(),comp_node);

	//Iterates through all nodes in node_list_vect to establish a matrix
of connections
	//Starts from nodes at the centre, and works it's way out.
	Node work_node;
	for(int i=0; i<num_nodes; i++)
	{
		work_node=node_list_vect.at(i);

		//Calculate distance from work node to all other nodes
		for (it=node_list_vect.begin(); it!=node_list_vect.end();
++it)
		{
			(*it).dist_fromnode=dist_between(work_node,(*it));
		}

		//Create truncated list that does not contain all previous
work nodes

	node_list_trunc.assign(node_list_vect.begin()+i+1,node_list_vect.end
());
		//print_node_vector(node_list_trunc);
```

```
            //cin<<endl<<"delay"<<a<<endl;

            //Sorting truncated list via the newly calculated distance

      sort(node_list_trunc.begin(),node_list_trunc.end(),comp_nodefromnode
);

            link_found=false;
            //Find all nodes within connec_within distance to working node
            for (it=node_list_trunc.begin(); it!=node_list_trunc.end();
++it)
            {
                  if((*it).dist_fromnode<connec_within)
                  {
                        //Create found link
                        Node_link temp_link;
                        temp_link.x1=work_node.x;
                        temp_link.y1=work_node.y;
                        temp_link.x2=(*it).x;
                        temp_link.y2=(*it).y;
                        temp_link.dist=sqrt(pow(temp_link.x1-
temp_link.x2,2)+pow(temp_link.y1-temp_link.y2,2));
                        //cout<<"temp link without method"<<endl;

      //cout<<temp_link.x1<<"\t"<<temp_link.y1<<"\t"<<temp_link.x2<<"\t"<<
temp_link.y2<<"\t"<<temp_link.dist<<endl;

                        //Add new link to link_list
                        link_list.push_back(temp_link);
                        //Set flag to indicate that a proximity link has
been found.
                        link_found=true;
                  }
            }

            //If there are no connecting nodes within a distance of
connec_within, then connect
            //to the closest node
            if(!link_found)
            {
                  calc_dist_fromNode(&node_list_temp,work_node);
                  //cout<<"work node"<<endl;
                  //cout<<work_node.x<<"\t"<<work_node.y<<endl;


      sort(node_list_temp.begin(),node_list_temp.end(),comp_nodefromnode);
                  //cout<<"Sorted temporary list"<<endl;
                  //print_node_vector(node_list_temp);
                  //cout<<"i: "<<i<<endl;

      for(it=node_list_temp.begin();it!=node_list_temp.end();it++)
                  {
                        //cout<<(*it).dist_fromnode<<endl;
                  }
                  cout<<"Short link added"<<endl;
                  Add_Link(&link_list,work_node,node_list_temp[1]);
                  //if(node_list_temp.at(3).dist_fromnode==0)
```

```
                //    cout<<"fuckup"<<endl;
                //Use node
                //connec_nodes.push_back(node_list_vect.at(1));
            }
        }
}

*/
void Node_handler::genLinkProx2(void)
{
      //Create vectors for node sorting
      vector<Node> node_list_near;
      vector<Node> node_list_temp;
      //node_list_temp.assign(node_list_vect.begin(),node_list_vect.end())
;

      //For iterating through node_list_vect
      vector<Node>::iterator it;
      vector<Node>::iterator near_it;
      Node_link temp_link;

      bool link_found=false;

      //vector<Node>::iterator to_delete;
      //Region delete_one;

      //Vector list of links between nodes
      //vector<int> connec_nodes;

      node_list_vect=node_Tracker.AllNodes();
      //Sorts node list by comparing distance from centre
      //node_list_vect doesn't change order after this
      sort(node_list_vect.begin(),node_list_vect.end(),comp_node);

      //Iterates through all nodes in node_list_vect to establish a matrix
of connections
      //Starts from nodes at the centre, and works it's way out.
      //Node work_node;
      //for(int i=0; i<num_nodes; i++)
      for(it=node_list_vect.begin();it!=node_list_vect.end();it++)
      {
            //Resetting link found flag
            link_found=false;
            //work_node=*it;
            node_list_near=node_Tracker.NodesNear((*it),connec_within*2);

            //Calculate distance from work node to all other nodes
            for (near_it=node_list_near.begin();
near_it!=node_list_near.end(); ++near_it)
            {
                  (*near_it).dist_fromnode=dist_between(*it,*near_it);
            }

            //Create truncated list that does not contain all previous
work nodes
```

```
      //node_list_trunc.assign(node_list_vect.begin()+i+1,node_list_vect.e
nd());
            //print_node_vector(node_list_trunc);

            //cin<<endl<<"delay"<<a<<endl;

            //Unnecessary sort
            //Sorting truncated list via the newly calculated distance

      //sort(node_list_near.begin(),node_list_near.end(),comp_nodefromnode
);


            //Find all nodes within connec_within distance to working node
            for (near_it=node_list_near.begin();
near_it!=node_list_near.end(); ++near_it)
            {
                  if((*near_it).dist_fromnode<connec_within)
                  {
                        //Create found link
                        if(near_it->dist_fromcentre>it->dist_fromcentre)
                        {
                              temp_link.x1=it->x;
                              temp_link.y1=it->y;
                              temp_link.x2=near_it->x;
                              temp_link.y2=near_it->y;
                              temp_link.dist=sqrt(pow(temp_link.x1-
temp_link.x2,2)+pow(temp_link.y1-temp_link.y2,2));
                              //cout<<"temp link without method"<<endl;

      //cout<<temp_link.x1<<"\t"<<temp_link.y1<<"\t"<<temp_link.x2<<"\t"<<
temp_link.y2<<"\t"<<temp_link.dist<<endl;

                              //Add new link to link_list
                              link_list.push_back(temp_link);
                              //Set flag to indicate that a proximity link
has been found.
                              link_found=true;
                        }


                  }
            }


            //delete_one=node_Tracker.AssignReg(it->x,it->y);

      //for(to_delete=node_Tracker.Ob_Map[delete_one].begin();to_delete!=n
ode_Tracker.Ob_Map[delete_one].end();to_delete++)
            //{
            //    if((*to_delete)==(*it))
            //    {
            //
      (node_Tracker.Ob_Map[delete_one]).erase(to_delete);
            //          break;
            //    }
```

```
            //}
            //to_delete=node_Tracker.Ob_Map[node_Tracker.AssignReg(it-
>x,it->y)];
            //if(node_Tracker.Ob_Map[node_Tracker.AssignReg(it->x,it-
>y)].size()<it->index)
            //    cout<<"Index too large
            //to_delete=to_delete+it->index;
            //Node gobyby;
            //gobyby=node_Tracker.Ob_Map[node_Tracker.AssignReg(it->x,it-
>y)].at((*it).index);
            //If there are no connecting nodes within a distance of
connec_within, then connect
            //to the closest node
            //if(!link_found)
            //if(false)
            //{
            //    calc_dist_fromNode(&node_list_temp,work_node);
            //    //cout<<"work node"<<endl;
            //    //cout<<work_node.x<<"\t"<<work_node.y<<endl;
            //
            //
      sort(node_list_temp.begin(),node_list_temp.end(),comp_nodefromnode);
            //    //cout<<"Sorted temporary list"<<endl;
            //    //print_node_vector(node_list_temp);
            //    //cout<<"i: "<<i<<endl;
            //
      for(it=node_list_temp.begin();it!=node_list_temp.end();it++)
            //    {
            //          //cout<<(*it).dist_fromnode<<endl;
            //    }
            //    Add_Link(&link_list,work_node,node_list_temp[1]);
            //    //if(node_list_temp.at(3).dist_fromnode==0)
            //    //    cout<<"fuckup"<<endl;
            //    //Use node
            //    //connec_nodes.push_back(node_list_vect.at(1));
            //}
            //node_Tracker.
      }
}
//Finds potential links between (x,y) and the other nodes in the node
field,
//then proceeds to check new links against all obstacles to check for
obstuctions.
//Finally, it adds links between it and the existing nodes to link_list.
void Node_handler::genLinkProx_NewNode(double x, double y)
{
      //Create vectors for node sorting
      vector<Node> node_list_near;

      //For iterating through node_list_vect
      vector<Node>::iterator near_it;
      Node_link temp_link;

      //Temporary holding container for links while they are checked for
obstructions
      vector<Node_link> temp_link_list;
```

```
	Node new_Node;
	new_Node.x=x;
	new_Node.y=y;

	node_list_near=node_Tracker.NodesNear(new_Node,connec_within*2);

	//Calculate distance from work node to all other nodes
	for (near_it=node_list_near.begin(); near_it!=node_list_near.end();
++near_it)
	{
		(*near_it).dist_fromnode=dist_between(new_Node,*near_it);
	}

	//Find all nodes within connec_within distance to working node
	for (near_it=node_list_near.begin(); near_it!=node_list_near.end();
++near_it)
	{
		if((*near_it).dist_fromnode<connec_within)
		{
			//Create found link
			temp_link.x1=new_Node.x;
			temp_link.y1=new_Node.y;
			temp_link.x2=near_it->x;
			temp_link.y2=near_it->y;
			temp_link.dist=sqrt(pow(temp_link.x1-
temp_link.x2,2)+pow(temp_link.y1-temp_link.y2,2));

			//Add new link to link_list
			temp_link_list.push_back(temp_link);

		}
	}

	//Now compare links in temp_link_list against all known obstacles.
	//What follows is a modified form of check_intercept.
	cout<<temp_link_list.size()<<" new being checked"<<endl;
	check_intercept(&temp_link_list);
	cout<<temp_link_list.size()<<" new links added"<<endl;

	//New links are added to the link_list
	vector<Node_link>::iterator add_link;
	Node temp_node1;
	Node temp_node2;
	for(add_link=temp_link_list.begin();add_link!=temp_link_list.end();
add_link++)
	{
		temp_node1.x=add_link->x1;
		temp_node1.y=add_link->y1;
		temp_node2.x=add_link->x2;
		temp_node2.y=add_link->y2;
		Add_Link(&link_list,temp_node1, temp_node2);

	}

}
//*****************************************************************
```

```
//Prints out vector of nodes to console display
//For debug purposes only
void Node_handler::print_node_vector(vector<Node> vector_out)
{
	vector<Node>::iterator it;
	for (it=vector_out.begin(); it!=vector_out.end();++it)
	{

	cout<<(*it).x<<'\t'<<(*it).y<<'\t'<<(*it).dist_fromcentre<<'\t'<<(*i
t).dist_fromnode<<'\t'<<endl;
	}
	cout<<endl;

}
//Prints out vector of nodes to file designated by *nodefile
void Node_handler::print_node_vector(char * nodefile)
{
	vector<Node>::iterator it;
	fstream outputnlist(nodefile,ios::out);

	for (it=node_list_vect.begin(); it!=node_list_vect.end();++it)
	{

	outputnlist<<(*it).x<<'\t'<<(*it).y<<'\t'<<(*it).dist_fromcentre<<en
dl; //'\t'<<(*it).dist_fromnode<<'\t'<<endl;
	}
	outputnlist.close();
}

//*****************************************************************
//prints out vector of links to console display
void Node_handler::print_link_list(vector<Node_link> links_out)
{
	vector<Node_link>::iterator it;
	for (it=links_out.begin(); it!=links_out.end(); ++it)
	{

	cout<<(*it).x1<<"\t"<<(*it).y1<<"\t"<<(*it).x2<<"\t"<<(*it).y2<<"\t"
<<(*it).dist<<endl;
	}
	cout<<endl;

}
//Prints out vector of links to file designated by *linnkfile
void Node_handler::print_link_list(char * linkfile)
{
	vector<Node_link>::iterator it;
	fstream outputllist(linkfile,ios::out);

	for (it=link_list.begin(); it!=link_list.end(); ++it)
	{

	outputllist<<(*it).x1<<"\t"<<(*it).y1<<"\t"<<(*it).x2<<"\t"<<(*it).y
2<<"\t"<<(*it).dist<<endl;
	}
```

```
	outputllist.close();
}
//*****************************************************************
//Calculates distance from all the vectors in Nodelist to myNode writes
the value to
//Nodelist[i].dist_fromnode for all valid i.
void Node_handler::calc_dist_fromNode(vector<Node> * Nodelist, Node
myNode)
{
	vector<Node>::iterator it;
	//cout<<"distances"<<endl;
	for (it=Nodelist->begin(); it!=Nodelist->end(); ++it)
	{
		(*it).dist_fromnode=sqrt(pow((*it).x-myNode.x,2)+pow((*it).y-
myNode.y,2));
		//cout<<(*it).dist_fromnode<<endl;
	}
	//cout<<"newly calucated"<<endl;
	//print_node_vector(*Nodelist);

}
//Concept test method.  Remove from finished product
//*****************************************************************
//Adds link to vlist using two Nodes.  Also calculates the distance
between them
void Node_handler::Add_Link(vector <Node_link> * vlist, Node origin, Node
dest)
{
	Node_link temp_link;
	temp_link.x1=origin.x;
	temp_link.y1=origin.y;
	temp_link.x2=dest.x;
	temp_link.y2=dest.y;
	temp_link.dist=sqrt(pow(temp_link.x1-
temp_link.x2,2)+pow(temp_link.y1-temp_link.y2,2));

	//Add new link to link_list
	vlist->push_back(temp_link);
}
//Enters successfully used nodes and links into memory
void Node_handler::add_Legacy(double x1, double y1, double x2, double y2)
{
	//Node_link temp_link;
	Node temp_node1;
	Node temp_node2;

	temp_node1.x=x1;
	temp_node1.y=y1;
	temp_node2.x=x2;
	temp_node2.y=y2;
	legacy_nodes.push_back(temp_node1);
	legacy_nodes.push_back(temp_node1);
	Add_Link(&legacy_links, temp_node1, temp_node2);
}
void Node_handler::integrate_LNodes(void)
{
	vector<Node>::iterator it;
```

```
	for(it=legacy_nodes.begin();it!=legacy_nodes.end();it++)
	{
		node_Tracker.Add_ob(*it);
		total_nodes++;
	}
}
void Node_handler::integrate_LLinks(void)
{
	vector<Node_link>::iterator it;
	Node temp_node1;
	Node temp_node2;
	for(it=legacy_links.begin();it!=legacy_links.end();it++)
	{
		temp_node1.x=it->x1;
		temp_node1.y=it->y1;
		temp_node2.x=it->x2;
		temp_node2.y=it->y2;
		Add_Link(&link_list,temp_node1, temp_node2);
	}
}
//Generates a random number of random obstacles
//For debugging purposes only
//void Node_handler::genRanObstacles(int num_obs)
//{
//	Node new_obstac;
//	for(int i=0; i<num_obs; i++)
//	{
//		new_obstac.x=(double)(rand()%(map_size_x*2*100) -
map_size_x*100)/100 + curr_robloc.x;
//		new_obstac.y=(double)(rand()%(map_size_y*2*100) -
map_size_y*100)/100 + curr_robloc.y;
//		ob_vect.push_back(new_obstac);
//	}
//}

void Node_handler::getObstacles(char * inObstFile)
{
	fstream obfileIn(inObstFile,ios::in);
	//ifstream obfileIn(inObstFile);
	//for(int i=0;i<100;i++)
	//string  xx;
	//string yy;
	Node new_obstac;
	double xx;
	double yy;
	//Try x and y as string, then use double atof(string) to convert.
	cout<<"Reading file: "<<inObstFile<<endl;
	//while(obfileIn>>xx)
	while(!obfileIn.eof())
	{
		//bool d=obfileIn.setf(std::ios::eof());
		//bool d=cin.setf(ios::eof);
		obfileIn>>xx;
		//cout<<"xx: "<<xx<<endl;

		//new_obstac.x=atof(xx.c_str());
		new_obstac.x=xx;
```

```
            obfileIn>>yy;
            //new_obstac.y=atof(yy.c_str());
            new_obstac.y=yy;
            ob_vect.push_back(new_obstac);
      }
      obfileIn.close();

}
void Node_handler::getTracker(ObstacleTracker * new_tracker)
{
      in_Tracker=new_tracker;
      //in_Tracker->PrintOb_Map();
}
void Node_handler::getTrackerNear(ObstacleTracker * new_tracker)
{
      nearOb_Tracker=new_tracker;
}
//Uses Dijkstra's algorithm to find the shortest route between curr_robloc
and destination
bool Node_handler::find_shortroute(void)
{

      HeapD<FHeap> heapF;
      Dijkstra *dijkstra;
      DGraph * myDgraph;

      double num_edges=0;
      double num_verts=0;
      //Array of distances from indexed vertices to source
      long * d;

      cout<<"Initializing myDgraph with "<<total_nodes<<" number of
nodes"<<endl;
      //Initialize graph to represent link_list
      myDgraph=new DGraph(total_nodes);

      //Filling created graph with links
      vector<Node_link>::iterator it;
      node_key.clear();
      int key_value=0;

      Node tempkey1;
      Node tempkey2;

      for(it=link_list.begin(); it!=link_list.end(); it++)
      //for(int i=0; i<30; i++)
      {
            //tempkey1=gen_NodeID(it->x1,it->y1);
            //tempkey2=gen_NodeID(it->x2,it->y2);
            tempkey1.x=it->x1;
            tempkey1.y=it->y1;
            tempkey2.x=it->x2;
            tempkey2.y=it->y2;
            //Keying first node from link
            //node_key.find(tempkey1);
            if(node_key.find(tempkey1)==node_key.end())
            {
```

```
                node_key[tempkey1]=key_value;
                key_value++;
            }
            //Keying second node from link
            if(node_key.find(tempkey2)==node_key.end())
            {
                node_key[tempkey2]=key_value;
                key_value++;
            }
            myDgraph->addNewEdge(node_key[tempkey1],node_key[tempkey2],it-
>dist);
            myDgraph->addNewEdge(node_key[tempkey2],node_key[tempkey1],it-
>dist);
            num_edges++;
      //it++;
      //cout<<"it: "<<it<<endl;
      }

      num_verts=node_key.size();
      cout<<"links per node: "<<(num_edges/num_verts)<<endl;

      cout<<"Instantiating dijkstra"<<endl;
      dijkstra= new Dijkstra(total_nodes,&heapF);


      d= new long[total_nodes];
      //cout<<"d instantiated"<<endl;
    for(long v = 0; v < total_nodes; v++) d[v] = INFINITE_DIST;  //
initialise
      //cout<<"D initialized to "<<INFINITE_DIST<<endl;

      dijkstra->init(myDgraph);
      cout<<"dijkstra intialized"<<endl;

      //Clears the outroute stack.
      while(!outroute.empty())
      {
            outroute.pop();
      }
      outroute=dijkstra-
>run(d,node_key[curr_robloc],node_key[destination]);
      cout<<"size outroute: "<<outroute.size()<<endl;
      //for(int k=0;k<total_nodes;k++)
      //{
      //    if(d[k]!=0)
      //          cout<<d[k]<<endl;
      //}

      //cout<<"d of destination: "<<d[node_key[destination]]<<endl;
      //cout<<"d of rob position: "<<d[node_key[curr_robloc]]<<endl;
      //if(d[node_key[curr_robloc]]==INFINITE_DIST)
      //if(outroute.empty())
      if(outroute.size()<=1)
      {
            //cout<<"d of destination: "<<d[node_key[destination]]<<endl;
            delete dijkstra;
            delete myDgraph;
```

```
            delete [] d;
            return false;
      }

      delete dijkstra;
    delete myDgraph;
      delete [] d;
      return true;

      //for( int i=0;i<30;i++)
      //{
      //    cout<<"d["<<i<<"]= "<<d[i]<<endl;
      //}


      //map<Node,int>::iterator iter;
      //Node * tempNode;
}

void Node_handler::print_shortroute(char * shortmapfile)
{
      fstream shmf(shortmapfile,ios::out);
      int key_num;
      map<Node,int>::iterator it;

      while(!outroute.empty())
      {
            if(outroute.empty())
            {
                  cout<<"outroute is empty"<<endl;
                  return;
            }
            key_num=outroute.top();

            outroute.pop();
            it=node_key.begin();

            while(((it->second)!=key_num)&&it!=node_key.end())
            {
                  //cout<<"Searched key_num: "<<it->second<<endl;
                  if(it==node_key.end())
                  {
                        cout<<"Error: Index number not found in short
list"<<endl;
                        return;

                  }
                  it++;
            }
            //cout<<(it->first).x<<"\t"<<(it->first).y<<"\tIndex number:
"<<key_num<<endl;
            shmf<<(it->first).x<<"\t"<<(it->first).y<<"\t"<<key_num<<endl;

      }
      //cout<<"Short route print finished"<<endl;
      shmf.close();
      return;
```

```
}

//Outputs the shortest route created by find_shortroute() in a
vector<Node>
vector<Node> Node_handler::out_shortroute(void)
{
	vector<Node> shortroute_v;
	map<Node,int>::iterator it;

	int temp_key;
	Node temp_node;

	while(!outroute.empty())
	{
		temp_key=outroute.top();
		outroute.pop();

		it=node_key.begin();
		while(((it->second)!=temp_key)&&it!=node_key.end()) it++;
		if(it==node_key.end())
			cout<<"Error: temp_key does not exist in node_key (node
map)"<<endl;
		temp_node.x=it->first.x;
		temp_node.y=it->first.y;
		shortroute_v.push_back(temp_node);
	}
	return shortroute_v;

}
void Node_handler::print_obstacles(void)
{
	vector<Node>::iterator it;
	for(it=ob_vect.begin(); it!=ob_vect.end(); it++)
	{
		cout<<(*it).x<<"\t"<<(*it).y<<endl;
	}
}
//Print obstacle list out to file defined by obfile
void Node_handler::print_obstacles(char * obfile)
{
	vector<Node>::iterator it;
	fstream outputolist(obfile,ios::out);

	for (it=ob_vect.begin(); it!=ob_vect.end(); ++it)
	{
		outputolist<<(*it).x<<"\t"<<(*it).y<<endl;
	}

	outputolist.close();
}
void Node_handler::print_legacy(char *lfile)
{
	vector<Node_link>::iterator it;
	fstream legacyout(lfile,ios::out);

	for(it=legacy_links.begin(); it!=legacy_links.end(); it++)
```

```
	{
		legacyout<<it->x1<<"\t"<<it->y1<<"\t"<<it->x2<<"\t"<<it-
>y2<<endl;
	}

	legacyout.close();
}
//checks the obstacle intercepts across the vector <Node_link> link_list
void Node_handler::check_intercept(void)
{
	vector <Node_link>::iterator link_it;
	vector <Node>::iterator ob_it;
	vector<Node> localNodes;

	bool delete_link=false;

	double lost_count=0;
	double linknum=0;
	double start_links=link_list.size();
	cout<<"Start of link loop"<<endl;

	Node temp_node;

	link_it=link_list.begin();

	while(link_it!=link_list.end())
	{
		delete_link=false;

		//Unnecessary value that tracks which link is being examined
		linknum++;
		temp_node.x=link_it->x1;
		temp_node.y=link_it->y1;
		localNodes=in_Tracker->NodesNear(temp_node,link_it-
>dist+R_radius);
		//Loops cycles through all obstacles vector ob_vect until it
finds one that blocks link_it
		//for(ob_it=ob_vect.begin(); ob_it!=ob_vect.end(); ob_it++)
		//if(localNodes.size()>0)
			//cout<<"Nodes actually being passed:
"<<localNodes.size()<<endl;
		for(ob_it=localNodes.begin(); ob_it!=localNodes.end();
ob_it++)
		{

	delete_link=check_intercept_single(&(*link_it),&(*ob_it),R_radius);
			if(delete_link)
				break;
		}
		if(delete_link)
		{
			lost_count++;
			link_it=link_list.erase(link_it);
		}
		else
		{
			link_it++;
```

```
            }
      }
      //cout<<"should trivial: "<<trive<<endl;
      //cout<<"actual Nontrivial: "<<nontrive<<endl;
      //cout<<"Actually deleted: "<<actuale<<endl;

      cout<<"Lost count: "<<lost_count<<endl;
      cout<<"Starting links: "<<start_links<<endl;
      cout<<"Ending links: "<<link_list.size()<<endl;
}

//Checks the obstacle intercepts across vector<Node_link> short_list
void Node_handler::check_intercept(vector<Node_link> * short_list)
{
      cout<<"Checking intercept"<<endl;

      vector <Node_link>::iterator link_it;     //Iterates through vector
of links in short_list
      vector <Node>::iterator ob_it;                  //iterates through
vector of obstacles
      vector <Node> localNodes;                       //Obstacles local to
the link under examination

      bool delete_link=false;                         //If true, a link is
obstructed and will be deleted

      double lost_count=0;
      double linknum=0;
      double start_links=link_list.size();

      Node temp_node;

      link_it=short_list->begin();

      //iterates through the links in short_list.
      while(link_it!=short_list->end())
      {
            delete_link=false;

            //Unnecessary value that tracks which link is being examined
            linknum++;

            //Finds nodes within dist+R_radius of Node temp_node
            temp_node.x=link_it->x1;
            temp_node.y=link_it->y1;
            localNodes=in_Tracker->NodesNear(temp_node,link_it-
>dist+R_radius);

            //Loops cycles through all obstacles vector ob_vect until it
finds one that blocks link_it
            for(ob_it=localNodes.begin(); ob_it!=localNodes.end();
ob_it++)
            {

      delete_link=check_intercept_single(&(*link_it),&(*ob_it),R_radius);
                  if(delete_link)
                        break;
```

```
            }
            if(delete_link)
            {
                  lost_count++;
                  link_it=short_list->erase(link_it);
            }
            else
            {
                  link_it++;
            }
      }

      cout<<"Lost count: "<<lost_count<<endl;
      cout<<"Starting links: "<<start_links<<endl;
      cout<<"Ending links: "<<link_list.size()<<endl;
}
//Checks the links in link_list for intercepts with the obstacles stored
in nearObs_Tracker,
//rather than in_Tracker
void Node_handler::check_intercept_newObs(void)
{
      vector <Node_link>::iterator link_it;
      vector <Node>::iterator ob_it;
      vector<Node> localNodes;

      bool delete_link=false;

      double lost_count=0;
      double linknum=0;
      double start_links=link_list.size();
      cout<<"Start of link loop"<<endl;

      Node temp_node;

      link_it=link_list.begin();

      while(link_it!=link_list.end())
      {
            delete_link=false;

            //Unnecessary value that tracks which link is being examined
            linknum++;
            temp_node.x=link_it->x1;
            temp_node.y=link_it->y1;

            //Finds all nodes within dist+R_radius of temp_node
            localNodes=nearOb_Tracker->NodesNear(temp_node,link_it-
>dist+R_radius);

            //Loops cycles through all obstacles vector ob_vect until it
finds one that blocks link_it
            for(ob_it=localNodes.begin(); ob_it!=localNodes.end();
ob_it++)
            {
                  delete_link=check_intercept_single(&(*link_it),
&(*ob_it),R_radius);
                  if(delete_link)
```

```
                        break;
                }
                if(delete_link)
                {
                        lost_count++;
                        link_it=link_list.erase(link_it);
                }
                else
                {
                        link_it++;
                }
        }

        cout<<"Lost count: "<<lost_count<<endl;
        cout<<"Starting links: "<<start_links<<endl;
        cout<<"Ending links: "<<link_list.size()<<endl;

}
bool Node_handler::check_intercept_single(Node_link * check_link, Node *
obstac, double range)
{
        //Distances from each examined link to the obstacle in question.
        double dis_end1;
        double dis_end2;

        //Non-trivial proximity check variables
        Node ab;                //Node that defines the vector of link
        double ab_mag=0;  //magnitude of vector defined by ab
        Node ab_unit;           //Unit vector of ab
        Node as;                //Vector from one end of the link to the
obstacle being checked

        double d1;              //distance from one end of the link to an
obstacle
        double d2;              //distance from the other end of the link to
an obstacle

        dis_end1=dist_between(obstac->x,obstac->y,check_link->x1,check_link-
>y1);
        dis_end2=dist_between(obstac->x,obstac->y,check_link->x2,check_link-
>y2);

        if((dis_end1<range)||(dis_end2<range))
        {
                //Link is obstructed, so return true;
                return true;
        }

        //Create variables for next phase of proximity check
        //relative vector of Link
        ab.x=check_link->x2-check_link->x1;
        ab.y=check_link->y2-check_link->y1;
        //Magnitude of the relative vector of Link
        ab_mag=sqrt(pow(ab.x,2)+pow(ab.y,2));
        //ab_mag=dist_between(ab.x, ab.y, 0, 0);
        //Unit vector of the relative vector of the Link
        ab_unit.x=ab.x/ab_mag;
```

```
	ab_unit.y=ab.y/ab_mag;

	//Creation of vector between one end of the link and the obstacle
	as.x=obstac->x-check_link->x1;
	as.y=obstac->y-check_link->y1;

	d1=as.x*ab_unit.x+as.y*ab_unit.y;

	if((d1>0)&&(d1<ab_mag))
	{
		//Cross product of as and ab.  Gets distance to line ab.
		d2=fabs(as.x*ab_unit.y-as.y*ab_unit.x);
		if (d2<range)
		{
			//link is obstructed, so return true;
			return true;
		}
	}
	return false;
}
//Returns true if the route designated by the input arguments is blocked.
bool Node_handler::route_blocked(double x1, double y1, double x2, double
y2)
{
	double dist=sqrt(pow(x1-x2,2)+pow(y1-y2,2));
	Node mid_node;
	vector<Node> localNodes;
	mid_node.x=(x1+x2)/2;
	mid_node.y=(y1+y2)/2;

	//double dis_end1, dis_end2, ab_mag;
	//Node ab, as, ab_unit;

	////bool delete_link;

	////Required distances
	//double d1, d2;
	//
	bool delete_link=false;

	localNodes=in_Tracker->NodesNear(mid_node,(dist/2)+R_radius);
	//If there are no obstacles nearby, then return false;
	if(localNodes.size()==0)
	{
		//cout<<"No obstacles nearby"<<endl;
		return false;
	}
	vector<Node>::iterator ob_it;
	for(ob_it=localNodes.begin(); ob_it!=localNodes.end(); ob_it++)
	{

	delete_link=check_intercept_single(&Node_link(x1,y1,x2,y2),
&(*ob_it),R_radius);
			if(delete_link)
				break;
	}
	if(delete_link)
```

```
	{
		cout<<endl;
		cout<<"Link: "<<x1<<"\t"<<y1<<"\t"<<x2<<"\t"<<y2<<endl;
		cout<<"blocked by: "<<ob_it->x<<"\t"<<ob_it->y<<endl;
		return true;
	}
	else
		return false;
}

double Node_handler::dist_between(double x1, double y1, double x2, double
y2)
{
	double distance=0;
	distance=sqrt(pow(x1-x2,2)+pow(y1-y2,2));
	return distance;
}
double Node_handler::dist_between(Node mynode1, Node mynode2)
{
	return dist_between(mynode1.x,mynode1.y, mynode2.x, mynode2.y);
}
```

APPENDIX D

ArnlEnv.cpp main source code

```
#include "Aria.h"
#include "Arnl.h"
#include "ArNetworking.h"
#include "time.h"
#include "math.h"
#include "ObstacleTracker.h"
#include "Node_handler.h"

#include <sstream>
#include <fstream>
#include <iostream>
#include "math.h"
#include "stdio.h"


using namespace::std;


void main(int argc, char** argv)
{

	//Aria initialization
	// mandatory init
	Aria::init();
	Arnl::init();

	//Naming obstacle storage files
	char * ObMapfile="ObstacleMap.txt";
	//Qualified obstacle list
	char * QObMapfile="QObstacleMapGarb.txt";
	//Naming log files
	//Mapped obstacles
	char * FullTest="mappedObs.txt";
	//Hypothetical traveled by robot
	char * legacyfile="LegacyList.txt";
	//Actual route traveled by robot
	char * actualr="Actualroute.txt";
	fstream actr(actualr,ios::out);

	//Tracks all obstacles
	ObstacleTracker ob_Tracker;
```

```
	ob_Tracker.setNoQ(false);
	ob_Tracker.setinRange(50);
	ob_Tracker.set_reg_size(2.1*50);

	//Tracks obstacles found since last route was found
	ObstacleTracker near_Tracker;
	near_Tracker.setNoQ(true);
	near_Tracker.setinRange(0);

	//Creates routes
	Node_handler myhandler;
	//Attaching trackers to myhandler
	myhandler.getTracker(&ob_Tracker);
	myhandler.getTrackerNear(&near_Tracker);
	//Setting the goal of the robot
	myhandler.set_destination(3000, 0000);
	myhandler.set_numnodes(2000);
	myhandler.set_radii(270);
	ArPose current_pose(0,0);

	//Poses used to attach legacy links
	ArPose leg_current_pose(current_pose);
	ArPose leg_previous_pose;

	bool finished=false;
	bool route_found=false;
	bool route_blocked=false;
	bool route_initialized=false;

	//Unused value
	double stall_limit=6000;
	//Keeps track of distance traveled
	DistanceTracker dTracker;

	//iterator to cycle through the vector containing the short route
	vector<Node>::iterator short_it;

	//Vector that contains any the generated short route any time it is
created
	vector<Node>  the_way;
	the_way.clear();


	//Creation of robot object.
	ArRobot robot;

	//Parsing input arguments
	ArArgumentParser parser(&argc, argv);
	//Creating simple connector to connect to robot
	ArSimpleConnector simpleConnector(&parser);

	parser.loadDefaultArguments();
	printf("Initialized\n");

	//Creation of a sonar object and attaching it to the ARrobot object
	ArSonarDevice sonar;
	robot.addRangeDevice(&sonar);
```

```
	printf("sonar device added\n");

	// Connect the robot
	if (!simpleConnector.connectRobot(&robot))
	{
		ArLog::log(ArLog::Normal, "Could not connect to robot...
exiting");
		Aria::exit(3);
	}

	//Confirm connection
	bool rcon=robot.isConnected();
	if(rcon)
	{
		printf("Connected\n");
	}
	if(!rcon)
	{
		printf("Not connected\n");
	}

	//Key handler instantiation.
	//Key handler makes it so pressing escape key causes robot
disconnect and
	//program shutdown
	ArKeyHandler keyHandler;
	Aria::setKeyHandler(&keyHandler);

	// Attach the key handler to a robot now, so that it actually gets
	// some processing time so it can work, this will also make escape
	// exit
	robot.attachKeyHandler(&keyHandler);
	printf("You may press escape to exit\n");

	//Data collection position instantiation
	ArPose origin(0,0,0);
	//ArPose endpos(1000,1000,90);

	//Creating gotoHead Action that causes robot to point a particular
direction
	//This doesn't seem to work due to non-functional robot.getX(),
.getY(), and .getTh()
	// methods.  This is a problem Aria has when running with some
libraries.
	//ArGotoHead gotoHead(180);

	//Actions used to make the robot wander for data collection and to
avoid collisions
	ArActionAvoidFront NoHitFront("avoid front obstacles",100, 200, 15);
	//Action used to make sure the robot turns before hitting a wall at
a oblique angle.
	//NoHitSide seems to interact badly with the algorithm.  Tends to
hijack the robots control
	//ArActionAvoidSide  NoHitSide("Avoid side",150, 5);
	//ArActionConstantVelocity ConstantVel("Constant Velocity",300);
	//Attaching the arAction that will be reused for travelling along
the discovered route
```

```
      ArActionGoto short_move("goto",origin);
      short_move.setCloseDist(100);
      //ArActionGotoStraight map_move("goto");

      //Creating the ArAction that will be reused for chasing moving
target
      //ArActionGoto target_move("goto",origin);
      //Action used to make sure the robot stops before hitting the target
it is chasing.
      ArActionLimiterForwards NoHitFrontStop("speed
limiter",300,300,500,1);

      //Attaching actions to robot with related priorities.
      //robot.addAction(&gotoPose, 100);
      //robot.addAction(&gotoHead, 50);
      //robot.addAction(&NoHitSide,90);
      robot.addAction(&NoHitFront,100);
      //robot.addAction(&ConstantVel,80);
      robot.addAction(&short_move,70);
      //robot.addAction(&NoHitSide,80);

      //Enabling motors.  Robot will not move otherwise.
      robot.enableMotors();

      //Creating SonarReading for each forward sonar unit
      ArSensorReading * frontSonar[8];
      for(int i =0;i<8;i++)
      {
            frontSonar[i]=robot.getSonarReading(i);
      }
      ArPose son_Ping;

      //Spinning the robot off on its own thread.
      robot.runAsync(true);

      //Keep track of whether or not the sonar data is new
      int oldCounter=0;
      int newCounter;

      bool SonIsNew=false;
      bool FirstGoal=true;

      bool first_loop=true;

      //Sets up value for time stall
      ArTime time_stall;
      time_stall.setToNow();
      //Robot's maximum translation speed, in mm/s
      double max_robot_speed=1200;
      //Limits robots maximum speed to 200mm/s
      robot.setTransVelMax(200);

    while((!finished)&&(Aria::getRunning()))
      {

            //Checking if Ranging data is new
            SonIsNew=true;
```

```
            robot.lock();
            newCounter=frontSonar[0]->getCounterTaken();
                robot.unlock();
            //If the counter has changed, the sonar data is considered
new
            if(newCounter!=oldCounter)
            {
                  SonIsNew=true;
            }
            else
            {
                  SonIsNew=false;
                  //robot.unlock();
            }

          if((SonIsNew) && (newCounter!=0))
          {
               //Cycling through the 8 sonar sensors
            for(int i=0;i<8;i++)
            {
                  if(frontSonar[i]->getRange()<3000)
                  {
                        robot.lock();
                        son_Ping=frontSonar[i]->getPose();
                        robot.unlock();
                        //Adding new sonar data to ObstacleTrackers
                        ob_Tracker.Add_ob(son_Ping.getX(),
son_Ping.getY());
                        near_Tracker.Add_ob(son_Ping.getX(),
son_Ping.getY());
                  }

            }
            oldCounter=newCounter;

          }

          //Updating current location
          robot.lock();
          current_pose=frontSonar[0]->getPoseTaken();
          robot.unlock();

          //Keeping track of how far the robot has traveled, and logging
it's current position

      dTracker.new_waypoint(current_pose.getX(),current_pose.getY());
          //Logging current location
          actr<<current_pose.getX()<<"\t"<<current_pose.getY()<<endl;

          ArPose temp_goal=short_move.getGoal();


          //Only check if route blocked if this is not the first time
through the main loop
          if(!first_loop)
          {
                //temp_goal, and leg_current_pose
```

```
                  //route_blocked=myhandler.route_blocked((short_it-1)->x,
(short_it-1)->y, temp_goal.getX(), temp_goal.getY());

      //route_blocked=myhandler.route_blocked(leg_current_pose.getX(),
leg_current_pose.getY(), temp_goal.getX(), temp_goal.getY());

      route_blocked=myhandler.route_blocked(current_pose.getX(),
current_pose.getY(), temp_goal.getX(), temp_goal.getY());
            }
            //Check if the robot has become lodged physically against a
wall by checking if it has moved
            //If a certain amount of time has passed without time_stall
being reset, then
            //dijkstra's is reinitiated
            //time_stall is reset whenever the robot arrives at a goal
point
            //stall_limit is how far the robot should have to travel in a
straight line between waypoints
            stall_limit=2*sqrt(pow(leg_current_pose.getX()-
temp_goal.getX(),2)+pow(leg_current_pose.getY()-temp_goal.getY(),2));
            //Checks if time_stall has reached its maximum
            //Value is multiplied by 5 in order to provide some leeway
            if(time_stall.secSince()>((stall_limit/max_robot_speed)*5))
            {
                  cout<<"Time stalled **********************"<<endl;
                  route_blocked=true;

            }

            //Check if the robot is stuck in a cicle by checking if the
robot has traveled much further than it should to get to it's goal
            if(dTracker.getLeg()>stall_limit)
            {
                  cout<<"Distance Stalled ************************"<<endl;
                  cout<<"leg: "<<dTracker.getLeg()<<endl;
                  cout<<"distance checked:
"<<sqrt(pow(leg_current_pose.getX()-
temp_goal.getX(),2)+pow(leg_current_pose.getY()-
temp_goal.getY(),2))<<endl;
                  route_blocked=true;
            }

            //Robot is stopped and the current location in myhandler is
updated
            if(route_blocked)
            {
                  cout<<"route blocked"<<endl;
                  robot.stop();

      myhandler.set_robloc(current_pose.getX(),current_pose.getY());

            }
            if(first_loop | route_blocked)
            {
                  //Short cycle route generation.
                  //Appends nodes and links to previously exisiting data
                  if(!first_loop)
```

```
                {
                        cout<<"Commencing short cycle route
generation"<<endl;

      myhandler.genLinkProx_NewNode(current_pose.getX(),current_pose.getY(
));
                        cout<<"Size of new obstacle list:
"<<near_Tracker.AllNodes().size()<<endl;;
                        myhandler.check_intercept_newObs();
                        route_found=myhandler.find_shortroute();
                        route_blocked=!route_found;
                }
                myhandler.set_numnodes(2000);
                //If a route cannot be found through short cycle route
generation, then myhandler
                //is reset and route finding begins from scratch
                while(first_loop | route_blocked)
                {
                        cout<<"Commencing long cycle route
generation"<<endl;
                        first_loop=false;
                        myhandler.auto_config();
                        myhandler.auto_connecDist();
                        cout<<"Generating Nodes"<<endl;
                        myhandler.genNodes_int(true);
                        //Adding known safe nodes
                        cout<<"Integrating LNodes"<<endl;
                        myhandler.integrate_LNodes();
                        //Creating links
                        cout<<"Linking"<<endl;
                        myhandler.genLinkProx2();
                        cout<<"Integrating LLinks"<<endl;
                        //Adding known safe links
                        myhandler.integrate_LLinks();
                        //Removing links obscured by obstacles
                        cout<<"Checking intecepts"<<endl;
                        myhandler.check_intercept();
                        //Applying dijkstra's to links in myhandler
                        route_found=myhandler.find_shortroute();
                        route_blocked=!route_found;

                        if(!route_found)
                        {
                              cout<<"Entering second phase path
finding"<<endl;
                              myhandler.auto_config(false);
                              myhandler.genNodes_int(false);
                              myhandler.genLinkProx2();
                              myhandler.check_intercept();
                              route_found=myhandler.find_shortroute();
                        }
                        route_blocked=!route_found;
                        if(route_found)
                              near_Tracker.clear();
                        //Increase node density in case of failure, and
repeat
                        if(route_blocked)
```

```
      myhandler.set_numnodes(myhandler.num_nodes+1000);
                  }
            }

            //Should only do this part the first time it is crossed after
the route generator loop
            if(route_found)
            {
                  the_way=myhandler.out_shortroute();
                  short_it=the_way.begin();
                  route_found=false;
                  //near_Tracker.clear();
            }


            //if the robot has arrived at a waypoint, or if a new route
was just produced
            if((short_move.haveAchievedGoal())||
(short_it==the_way.begin()))
            {
                  //Reset the circle stall value
                  dTracker.resetLeg();
                  //Update current robot position
                  robot.lock();
                  current_pose=frontSonar[0]->getPoseTaken();
                  robot.unlock();
                  //Add just traversed link to the list of known safe
links
                  leg_previous_pose=leg_current_pose;
                  leg_current_pose=current_pose;

      myhandler.add_Legacy(leg_previous_pose.getX(),leg_previous_pose.getY
(),leg_current_pose.getX(), leg_current_pose.getY());

                  short_it++;
                  if(short_it!=the_way.end())
                  {
                        //Lookahead routine.  It lets the robot combine a
series of short links into
                        //a single longer one.  This produces smoother
movement and less chance of
                        //circle stalling
                        short_it++;

      while((short_it!=the_way.end())&&(!myhandler.route_blocked(current_p
ose.getX(), current_pose.getY(), short_it->x,short_it->y)))
                        {
                              short_it++;
                              cout<<"Skip"<<"\t";
                        }
                        cout<<endl;
                        short_it--;

                        //Set new local goal for hte robot
                        robot.lock();
```

```
                        short_move.setGoal(ArPose(short_it->x,short_it-
>y));
                        cout<<"New goal: "<<short_it->x<<"\t"<<short_it-
>y<<endl;
                        //Resume robot motion
                        robot.clearDirectMotion();
                        robot.unlock();
                        //Reset time_stall value
                        time_stall.setToNow();
                  }

                  if(short_it==the_way.end())
                        finished=true;
            }
      }

      //print logs of nodes and obstacles to txt files
      myhandler.print_legacy(legacyfile);
      ob_Tracker.listToFile(FullTest);
      actr.close();
      cout<<"Logs printed to file"<<endl;
      //Shut down Aria
      Aria::shutdown();
      Aria::exit(0);
      Aria::uninit();
}
```

REFERENCES

[1] Shortest Path Algorithms and Priority Queues in C++
University of Canterbury, Computer Science and Software Engineering website
Software by Shane Saunders
http://www.cosc.canterbury.ac.nz/research/RG/alg/spalg.html

[2] Mobile Robots by ActivMedia Robotics
P3-DX specification page – physical parameters comparable to obsolete P2 model
http://robots.mobilerobots.com/

[3] Mobile Robot Start-Up Positioning using Interval Analysis
A. B. Martínez, J. Escoda, A. Benedico;
Departament d'Enginyeria de Sistemes, Universitat Politècnica de Catalunya (ESAII-UPC)
E-mail: {Antonio.B.Martinez,Josep.Escoda,Toni.Benedico}@upc.edu

[4] Using Laser and Vision to Locate a Robot in an Industrial Environment: A Practical Experience
Alenya, G.; Escoda, J.; Martinez, A.B.; Torras, C.;
Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on
18-22 April 2005 Page(s):3528 – 3533

[5] Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths
M.L. Fredman, D. E. Willard
Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on
22-24 Oct. 1990 Page(s):719 - 725 vol.2

[6] An evaluation of concurrent priority queue algorithms
Qin Huang; Weihl, W.E.;
Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on
2-5 Dec. 1991 Page(s):518 – 525

[7] Fast data structures for shortest path routing: a comparative evaluation
Oberhauser, G.; Simha, R.;
Communications, 1995. ICC 95 Seattle, Gateway to Globalization, 1995 IEEE International Conference on
Volume 3, 18-22 June 1995 Page(s):1597 - 1601 vol.3

[8] A method for the shortest path search by extended Dijkstra algorithm
Noto, M.; Sato, H.;
Systems, Man, and Cybernetics, 2000 IEEE International Conference on
Volume 3, 8-11 Oct. 2000 Page(s):2316 - 2320 vol.3

[9] A New Shortest Path Algorithm based on Heuristic Strategy
Chen Xi; Fei Qi; Li Wei;
Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on
Volume 1, 2006 Page(s):2531 – 2536

[10] A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices
Barbehenn, M.;
Computers, IEEE Transactions on
Volume 47, Issue 2, Feb. 1998 Page(s):263

[11] Implementation of efficient algorithms for globally optimal trajectories
Polymenakos, L.C.; Bertsekas, D.P.; Tsitsiklis, J.N.;
Automatic Control, IEEE Transactions on
Volume 43, Issue 2, Feb. 1998 Page(s):278 – 283

[12] Shortest path planning on topographical maps
Saab, Y.; VanPutte, M.;
Systems, Man and Cybernetics, Part A, IEEE Transactions on
Volume 29, Issue 1, Jan. 1999 Page(s):139 – 150

[13] New Local Path Replanning Algorithm for Unmanned Combat Air Vehicle
Qinkun Xiao; Xiaoguang Gao; Xiaowei Fu; Haiyun Wang;
Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on
Volume 1, 2006 Page(s):4033 – 4037

[14] Fast unmanned vehicles task allocation with moving targets
Turra, D.; Pollini, L.; Innocenti, M.;
Decision and Control, 2004. CDC. 43rd IEEE Conference on
Volume 4, 14-17 Dec. 2004 Page(s):4280 - 4285 Vol.4

[15] On the heuristics of A* or A algorithm in ITS and robot path-planning
Goto, T.; Kosaka, T.; Noborio, H.;
Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on
Volume 2, 27-31 Oct. 2003 Page(s):1159 - 1166 vol.2

[16] Undirected single source shortest paths in linear time
Thorup, M.;
Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on
20-22 Oct. 1997 Page(s):12 – 21

[17] Fast data structures for shortest path routing: a comparative evaluation
Oberhauser, G.; Simha, R.;
Communications, 1995. ICC 95 Seattle, Gateway to Globalization, 1995 IEEE International Conference on
Volume 3, 18-22 June 1995 Page(s):1597 - 1601 vol.3

[18] The weighted Voronoi diagram and its applications in least-risk motion planning
Meng, A.C.-C.; Brooks, G.S.; Vermeer, P.J.;
Computers and Communications, 1989. Conference Proceedings., Eighth Annual International Phoenix Conference on
22-24 March 1989 Page(s):562 - 566

[19] Multi-constrained routing based on simulated annealing
Yong Cui; Ke Xu; Jianping Wu; Zhongchao Yu; Youjian Zhao;
Communications, 2003. ICC '03. IEEE International Conference on
Volume 3, 11-15 May 2003 Page(s):1718 - 1722 vol.3

[20] A New Shortest Path Algorithm based on Heuristic Strategy
Chen Xi; Fei Qi; Li Wei;
Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on
Volume 1, 2006 Page(s):2531 - 2536
Digital Object Identifier 10.1109/WCICA.2006.1712818

[21] On the heuristics of A* or A algorithm in ITS and robot path-planning
Goto, T.; Kosaka, T.; Noborio, H.;
Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on
Volume 2, 27-31 Oct. 2003 Page(s):1159 - 1166 vol.2

[22] A dynamic multi-source Dijkstra's algorithm for vehicle routing
Eklund, P.W.; Kirkby, S.; Pollitt, S.;
Intelligent Information Systems, 1996., Australian and New Zealand Conference on
18-20 Nov. 1996 Page(s):329 - 333
Digital Object Identifier 10.1109/ANZIIS.1996.573976

[23] Multi-constrained routing based on simulated annealing
Yong Cui; Ke Xu; Jianping Wu; Zhongchao Yu; Youjian Zhao;
Communications, 2003. ICC '03. IEEE International Conference on
Volume 3, 11-15 May 2003 Page(s):1718 - 1722 vol.3

[24] Reactive obstacle avoidance for mobile robots that operate in confined 3D workspaces
Vikerimark, D.; Minguez, J.;
Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean
16-19 May 2006 Page(s):1246 – 1251

[25] Modeling the Static and the Dynamic Parts of the Environment to Improve Sensor-based Navigation
Montesano, L.; Minguez, J.; Montano, L.;
Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on
18-22 April 2005 Page(s):4556 - 4562

[25] Global nearness diagram navigation (GND)
Minguez, J.; Montano, L.; Simeon, T.; Alami, R.;
Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on
Volume 1, 2001 Page(s):33 - 39 vol.1

[27] Nearness diagram navigation (ND): a new real time collision avoidance approach
Minguez, J.; Montano, L.;
Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on
Volume 3, 31 Oct.-5 Nov. 2000 Page(s):2094 - 2100 vol.3

[28] The obstacle-restriction method for robot obstacle avoidance in difficult environments
Minguez, J.;
Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on
2-6 Aug. 2005 Page(s):2284 - 2290

[29] Integration of planning and reactive obstacle avoidance in autonomous sensor-based navigation
Minguez, J.;
Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on
2-6 Aug. 2005 Page(s):2486 - 2492

[30] A "divide and conquer" strategy based on situations to achieve reactive collision avoidance in troublesome scenarios
Minguez, J.; Osuna, J.; Montano, L.;
Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on
Volume 4, Apr 26-May 1, 2004 Page(s):3855 - 3862 Vol.4

[31] Reactive Navigation for Non-holonomic Robots using the Ego-Kinematic Space
J. Minguez, L. Montano, J. Santos-Victor
Proceedings of the 2002 IEEE International Conference on Robotics and Automation, May 2002
Email: {jminguez,montano}@posta.unizar.es, jasv@isr.ist.utl.pt

[32] The ego-kinodynamic space: collision avoidance for any shape mobile robots with kinematic and dynamic constraints
Minguez, J.; Montano, L.;
Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on
Volume 1, 27-31 Oct. 2003 Page(s):637 - 643 vol.1

[33] Reactive navigation for non-holonomic robots using the ego-kinematic space
Minguez, J.; Montano, L.; Santos-Victor, J.;
Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on
Volume 3, 11-15 May 2002 Page(s):3074 – 3080

[34] A motion control of a two-wheeled mobile robot
Tsuchiya, K.; Urakubo, T.; Tsujita, K.;
Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on
Volume 5, 12-15 Oct. 1999 Page(s):690 - 696 vol.5

[35] Adaptive tracking control of a nonholonomic mobile robot
Fukao, T.; Nakagawa, H.; Adachi, N.;
Robotics and Automation, IEEE Transactions on
Volume 16, Issue 5, Oct. 2000 Page(s):609 – 615

[36] Path planning problems and solutions
Goldman, J.A.;
Aerospace and Electronics Conference, 1994. NAECON 1994., Proceedings of the IEEE 1994 National
23-27 May 1994 Page(s):105 - 108 vol.1

[37] An Evolutionary Artificial Potential Field Algorithm for Dynamic Path Planning of Mobile Robot
Cao Qixin; Huang Yanwen; Zhou Jingliang;
Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on
Oct. 2006 Page(s):3331 – 3336

[40] Cognitive based adaptive path planning algorithm for autonomous robotic vehicles
Razavian, A.A.; Sun, J.;
SoutheastCon, 2005. Proceedings. IEEE
8-10 April 2005 Page(s):153 – 160

[41] Support Vector Path Planning
Miura, J.;
Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on
Oct. 2006 Page(s):2894 – 2899

[42] Sensor-based path-planning algorithms for a nonholonomic mobile robot
Noborio, H.; Yamamoto, I.; Komaki, T.;
Intelligent Robots and Systems, 2000. (IROS 2000). Proceedings. 2000 IEEE/RSJ International Conference on
Volume 2, 31 Oct.-5 Nov. 2000 Page(s):917 - 924 vol.2