Master Thesis
Computer Science
Thesis no: MCS-2008:35
August 2008

Performance Comparison of AI Algorithms

Anytime Algorithms

Rehman Tariq Butt

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
SE – 372 25 Ronneby
Box 520
Sweden

Contact Information:

Author:	Rehman Tariq Butt
Tel No.:	+46737401917
E-mail:	rehmanatwork@hotmail.com

University advisor:
Stefan Johansson
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet	: www.bth.se/tek
Phone	: +46 457 38 50 00
Fax	: + 46 457 102 45

ABSTRACT

Commercial computer gaming is a large growing industry, that already has its major contributions in the entertainment industry of the world. One of the most important among different types of computer games are Real Time Strategy (RTS) based games. RTS games are considered being the major research subject for Artificial Intelligence (AI). But still the performance of AI in these games is poor by human standards because of some broad sets of problems. Some of these problems have been solved with the advent of an open real time research platform, named as ORTS. However there still exist some fundamental AI problems that require more research to be better solved for the RTS games. There also exist some AI algorithms that can help us solve these AI problems.

Anytime- Algorithms (AA) are algorithms those can optimize their memory and time resources and are considered best for the RTS games. We believe that by making AI algorithms anytime we can optimize their behavior to better solve the AI problems for the RTS games. Although many anytime algorithms are available to solve various kinds of AI problems, but according to our research no such study is been done to compare the performances of different anytime algorithms for each AI problem in RTS games. This study will take care of that by building our own research platform specifically design for comparing performances of selected anytime algorithms for an AI problem

Keywords: Artificial Intelligence (AI), Real Time Strategy (RTS) Games, AI Algorithms, AI Problems, Anytime Algorithms, A – Star, RBFS, Potential Fields, Path Finding, ORTS platform, PFPC platform etc

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This study may have never been completed without the support, motivation, love and believe from my friends and family; that guides me through difficult stages of my study. Thank you, All!

Especially thanks to:

- My parents, though being so far away but still support me with all means possible.
- Stefan Johansson, due to his good advice and guidance throughout the study that help me to better understand and perform this study.
- All my friends, family members and people around me, those support me through good and bad times.

CHAPTER 1: INTRODUCTION

This chapter as the name suggests will cover up the initial stages of this study. In Section 1.1, we will explain our problem in hand and also provide its background. In Section 1.2, we will define our purpose of the study and what objectives we want to achieve with it. Then in Section 1.3, we will list down our research questions that we want to answer; and towards the end in Section 1.4, we will define our research methodology to solve the problem in a systematic way.

1.1 Background

Commercial computer gaming is a large growing industry, that already has its major contributions in the entertainment industry of the world. Today these computer games are holding a multi- billion dollar industry. Despite this multi- billion dollar enterprise, there exists a relatively small set of different game genres [1, 3]. The most important among these game genres is Real- Time Strategy (RTS) - based games. RTS games are important, firstly because of their usage in simulations, especially for modern military training and secondly, games like Starcraft, Warcraft, and Age of Empire etc have sold millions of copies and earned billions of dollars, are getting ever more popular [1].

Artificial Intelligence (AI) is a field of science that deals with intelligence. It's a field of science that helps us to understand and build intelligent systems [2]. Although it's a relatively new science, but the existence of its intelligent systems is important in many fields of life, like in military, in medical for disease diagnostics, in robotics, in mathematics for proving theorems and of- course also in computer gaming [6]. Especially computer gaming is consider to be a major research subject for AI. Its importance in this area is such that a whole new field of AI named as 'Game- AI' came into existence, which deals specifically with making gaming logics [4].

Although commercial computer gaming is a major research subject for AI, its performance in this subject is not up to mark, and computer games still provide major challenges to the researchers of AI. RTS gaming, which has an extra importance due to its usage in simulations, the AI performance in these games is still poor by human standards [5]. This is because of the three broad reasons. Firstly, up- till now games are being launched by gaming companies, those spent most of their time on improving the game's graphics rather than the game's AI. Secondly, lack of AI competitions in this discipline deprives AI researchers with an opportunity to test their algorithms against each others. Thirdly, RTS games, as the name suggests have some severe time and memory constraints and AI algorithms have to perform real time to match these constraints [1]. First two of these three problems are very much solved with the advent of an open real time research platform, named as ORTS.

Open Real Time Strategy (ORTS) is a RTS research platform for the AI researchers to work on. ORTS, has many advantages that other commercial RTS platforms do not. Firstly, ORTS is a free of license and a free of cost software, which means that its source files are available freely and researchers can modify them depending upon their own game specifications. It's an open and expandable RTS platform as compare to close and not expandable commercial RTS platforms. Secondly, ORTS implements client- server network technology as compare to peer- to- peer network technology of other commercial RTS platforms. Client- Server technology provides us with additional advantages like, stopping map- revealing hacks and allowing users to connect through whatever user software they like [5]. Although ORTS now provide us with suitable research platform there still exists many fundamental RTS related AI problems [1].

Now a day, more and more research is being done to improve the performance of AI in RTS games. RTS games offered many fundamental AI related problems. Problems like Planning, Decision Making under Uncertainty, Learning, Reasoning, Resource Management, Collaboration and Path- Finding [1, 6]. There also exists many AI related algorithms like A* search, Iterative Deepening A* (IDA*), Recursive Best First Search (RBFS), Influence Diagrams, Potential Fields (PF), Genetic Algorithms (GA), Neural Network, Decision Trees etc, that can help us to solve these AI problems [6]. However, still more research is required to better understand these problems and algorithms, in order to improve the performance of AI in time and memory constrained environments of RTS games.

Anytime- Algorithms (AA) are the type of algorithms that provide intelligent systems with the capability to consume their execution time for better quality of results. The word anytime is used because unlike normal algorithms, we can stop these algorithms at anytime and expect them to return an output [7, 8]. However in order to declare an algorithm anytime, that algorithm has to satisfy certain properties. Properties like, Measurable quality, Mono-tonicity, Consistency, Diminishing returns, Interrupt-ability and Preempt-ability. As many computational tasks in RTS gaming are too complicated to be completed at real- time speeds, AA can help intelligent systems to intelligently allocate their computational time resources in the most effective way [7, 8]. We believe that by making above mentioned AI algorithms anytime we can better understand and optimized their behavior under RTS gaming environment.

Summarizing everything towards the end, we believe that RTS gaming is an important game genre, but the performance of AI in this genre is not up to mark because of some broad sets of problems. Some of these problems are being solved with the advent of a RTS research platform ORTS. Other problems can be better solved by making AI algorithms, anytime. Although many anytime algorithms are available to solve various kinds of AI problems, but according to our research no such study is been done to compare the performances of different anytime algorithms for each AI problem. This study will take care of that by building our own platform specifically for comparing performances of selected anytime algorithms for an AI problem.

1.2 Purpose & Objectives

The purpose of this study is to understand and analyze various AI related problems and algorithms in RTS gaming, and then comparing performances of the selected algorithms, after making these algorithms anytime, and by building our own platform. By comparing performances of the selected anytime algorithms we can conclude a decision that which algorithm has performed better for which problem.

This purpose would be achieved by using the following objectives:

- Identifying different AI relating problems and algorithms offered in RTS games.
- Analyzing anytime algorithms as a possible solution for the above discovered problems.
- Develop a solution for the above discovered problems by comparing performances of different algorithms, after making these algorithms anytime and by using our own platform.

1.3 Research Questions

The basic research questions regarding this study are listed below:

- What are the AI relating problems and algorithms offered in RTS games?
- Are anytime algorithms a possible solution for the discovered problems?
- How can we develop a solution for the discovered problems by comparing performances of different anytime algorithms and by using our own platform?
- Is the developed solution, a good enough solution for these AI problems?

1.4 Research Methodology

Firstly, a thorough literature review using books, articles, forums and journals has been conducted. The main purpose of this review is to answer some of the research questions that we are facing. Questions like what are the AI problems, and what AI algorithms are available to solve them in RTS games? ; What are anytime algorithms and are they provide a possible solution for AI problems?

Secondly for our experimentation part, a platform has been developed to compare the performances of different anytime algorithms. After making the comparison using different scenarios we made a decision that which anytime algorithm is better for which AI problem.

1.5 Outline

This paper is divided into 8 chapters. The first two chapters concern our theoretical phase and are result of our thorough literature review. Chapter 1 provides the background, purpose and objectives of this study. In Chapter 2 we will identify and analyze different AI problems and algorithms, and will select some for our experimentation. Then in Chapter 3, 4 and 5 we will discuss these selected AI algorithms one by one and convert them into anytime AI algorithms. In Chapter 6 we will discuss our specially build platform that we will use to conduct our experiments. In Chapter 7 we will conduct our experiments followed by the discussion about their results in Chapter 8.

CHAPTER 2: THEORETICAL STUDY

This chapter of theoretical study is a result of a thorough literature review that is been carried out to understand and solve some of our research questions, and to provide us with a solid ground to conduct our experiments later on. In Section 2.1, we will briefly discuss AI importance and performance in RTS games and then in Section 2.2, we will analyze and discuss different AI problems and algorithms, and select some for our experimentation. Towards the end in Section 2.3, we will briefly discuss anytime algorithms.

2.1 AI Importance and Performance in RTS Games

Artificial Intelligence (AI) is a field of science that deals with intelligence. It is difficult to define AI in any formal definition, because definitions of AI can vary along different dimensions. But in its simplistic form we can say that AI is a field of science that helps us to understand and build intelligent systems [6, 2]. In this section however we will discuss the importance and performance of AI in RTS games.

2.1.1 AI Importance in RTS Games

AI currently features in various fields of life, but computer gaming especially is considered to be a major research subject for AI. Its importance in this area is such that a whole new field of AI named as ‘Game- AI’ came into existence, which deals specifically with making gaming logics. Game- AI is now helping game developers in makings of characters, that can behave much alike humans and providing much more fun to play against [4]. Unfortunately, up till now most of the resources are consumed on improving the game’s graphics rather than game’s AI, but vast advancements in game’s hardware make it possible for game developers to assign more resources for Game- AI related computations [1, 11].

2.1.2 AI Performance in RTS Games

Although commercial computer gaming is a major research subject for AI, and its existence in computer gaming is important for their success, but still its performance in this area is not up to mark, and computer games still provide major challenges to the researchers of AI. Current performance of AI in these games is poor by human standards. This is because of the three broad reasons [5, 1].

2.1.2.1 Computer Gaming Companies

Up till now games are being launched by gaming companies, those spent more time on improving the game’s graphics rather than the game’s AI. It is been estimated that only 15% of the CPU time is dedicated for AI related tasks. But now vast advancements in hardware industry are allowing companies to assign more time for AI related tasks. Also the gaming companies are not willing to release their communication protocols or to allow AI researches to attach their AI modules to their products, which is necessary to allow researchers to test their AI algorithms [1].

2.1.2.2 Lack of AI Competitions

Lack of many AI competitions also deprives AI researchers with an opportunity to test their algorithms against each others. Currently most of the research is being done in research centers build up by computer gaming companies [1, 11].

2.1.2.3 Real- Time Constraints

Normally, most of the AI algorithms need large amount of computational time and memory to solve any given problem, but in RTS games there exists some severe time and memory constraints, and AI algorithms have to perform real time to match these constraints [1].

First two of these three problems are very much solved with the advent of a research platform, named as ORTS. It is a RTS research platform for the AI researchers to work on, and was first launched in 2001. It is a programming environment that allows AI researchers to conduct their AI experiments. ORTS provided us with two main advantages, firstly now we don't have to depend upon computer gaming companies, which are not willing to release their communication protocols to allow any external AI module to be attached to their products. Secondly now a tournament is been held every year, where different teams can participate and test their AI algorithms against each others in four different disciplines [1, 5].

2.2 AI Problems and Algorithms in RTS Games

RTS games offered many fundamental AI related problems like Planning, Decision Making under Uncertainty, Learning, Reasoning, Resource Management, Collaboration, Path- Finding etc. However there exists many more AI problems, but we are considering only those related to RTS games. There also exists many AI related algorithms that can help us to solve these problems. Algorithms like A* search, Iterative Deepening A* (IDA*), Recursive Best First Search (RBFS), Influence Diagrams, Potential Fields (PF), Genetic Algorithms (GA), Neural Network, Decision Trees etc [1, 6]. In this section we will discuss these problems and algorithms in brief.

2.2.1 AI Problems in RTS Games

Listed below are the AI problems in RTS gaming environment.

2.2.1.1 Planning

Planning in RTS games is an alternative to the most commonly used techniques (Scripts, Finite State Machines etc), for modeling a gaming character behavior. These techniques generally resulted in a static character behavior which in return fails to cope with dynamic environments of RTS games. The character in RTS games has to plan dynamically and well in advance, but as the RTS environment is so hostile and ever changing, it is difficult to achieve [12, 1].

2.2.1.2 Decision Making under Uncertainty

Decision making is considered to be an important feature in RTS games. A gaming character has to make many decisions during its game play. The quality of its decisions depends upon the quality and validity of its information about the RTS environments. But unfortunately, the character almost never has access to the whole information about RTS environments. This information can be of any kind such as enemy locations, base and ammo locations etc. The character has to make most decisions while uncertain, which makes it a difficult AI problem to solve [6, 1].

2.2.1.3 Learning

Learning can be explained as a process through which a gaming character can improve its behavior through the study of its own environment. The information about the RTS environments can not only be used for decision making but also for learning purposes. Learning can help the character by not repeating its mistakes and also to overcome its weaknesses. The rate with which the character can learn also matters, because we know that a human player needs only a handful of games to learn. Learning in RTS environments still requires lots of research [6, 1].

2.2.1.4 Reasoning

Reasoning can be defined as a process through which a gaming character can use its information of the environment to form some meaningful representation of it. With the help of this meaningful representation, the character can correctly deduce what action to take next. In RTS gaming, reasoning can be used to represent terrain in some meaningful format like passable or impassable sections, which can be very helpful in making correct decisions. But currently most AI programs normally ignore these issues and their performance in this regard is poor by human standards [6, 1].

2.2.1.5 Resource Management

Unlike other computer games genres, resource management has a central importance in RTS gaming. A gaming character can gather many different kinds of resources during a game play, like ammo, health, money etc. The way the character uses these resources is very important and crucial for its better performance in the RTS games. An example can be of the character, which prefers health resource, and can easily run out of money and ammo thus bound to dead. So a gaming character has to intelligently collect and consume its resources to be successful in RTS games [1].

2.2.1.6 Collaboration

RTS gaming environments generally contain large number of gaming characters; those can possibly be divided into groups, and competing against each other. In such cases, these gaming characters have to work as a team to perform better than the opponent group. In scenario like this, collaboration and communication between the group members gains central importance. The performance of human players working as a group is much better than those of the computer characters. Further research is required to improve and better optimize their working as a group [1, 6].

2.2.1.7 Path- Finding

Path finding in its simplest can be explained as a process to find shortest route between two given points in a RTS environment. This is the single most important AI problem which gains much more attention then any other AI problem. But still its performance is not satisfactory. The big reason is that, RTS environments have some critical memory and time constraints and this process of path finding requires lots of computational resource. Although the performance of graphics hardware accelerators is increasing but still calculating optimal paths for many gaming characters in RTS environments require more research in this area [1].

2.2.1.8 AI Problem Selection for our Experiment

After thorough literature review of different AI problems in RTS gaming environments, we believe that path finding is one of the most important AI problem. Although lots of research is currently been carried out on this, but still more research is required to better optimize its performance in RTS environments. So we have selected path finding, as an AI problem for our experiments later on.

2.2.2 AI Algorithms in RTS Games

The AI algorithms that can help us to solve the above selected problem are large in number and it is difficult for us to discuss all of them in here. However we have categorized all these algorithms into four broad search categories and will discuss them briefly.

2.2.2.1 Un- Informed or Non- Heuristic Search

This category of uninformed or non- heuristic search contains algorithms those do not have any additional information about their environment apart from that already been provided in the problem definition. Problem definition contains information about initial state, state space, successor function, goal test and path cost. Apart form that no additional information is been provided. This category is also known as blind search category [6].

Algorithms available in this category are listed below:

- Breath- first Search
- Depth- first Search
- Depth- limited Search
- Iterative Deepening Depth- first Search
- Bidirectional Search

2.2.2.2 Informed or Heuristic Search

This category of informed or heuristic search contains algorithms those have some additional problem specific information about their environments apart form that already been provided in the problem definition. With this additional information these algorithms can make a comparison between two different states and possibly select the suitable one. This comparison between two states is made with the help of a special function named as, evaluation function. These algorithms generally perform much better than uninformed search algorithms and are widely used for solving various AI problems [6].

Some of the algorithms available in this category are listed below:

- Greedy Best- first Search
- A* Search
- Iterative Deepening A* Search
- Recursive Best First Search

2.2.2.3 Local Search Algorithms

The two types of search categories that we have discussed so far need to explore search space systematically. This is achieved by keeping track of some or all of the states that are been explored so far and when a goal is found the path from the initial state to the goal state becomes the solution of that problem. But there also exists such kind of problems in which path to the goal does not matter, and all that matter is the goal state. Such kind of problems is being solved by using algorithms of local search category. This search category has two main advantages over the above discussed, firstly this category has little or no memory consumptions and secondly it is suitable for finding solutions of problems those have infinite search space, for which systematic searching is not suitable [6].

Some of the algorithms available in this category are listed below:

- Hill- climbing Search
- Taboo Search
- Simulated Annealing Search
- Local Beam Search
- Genetic Algorithm

2.2.2.4 Some Other Algorithms

List of some other important AI algorithms are listed below:

- Neural Networks
- Decision Trees
- Influence Diagrams
- Potential Fields

2.2.2.5 AI Algorithms Selection for our Experiment

In order to select appropriate algorithms to conduct our experiment, we firstly need to analyze the environments of RTS games. RTS gaming environments are generally consists upon finite number of states, and also some problem specific information is available along with problem definition. With these things available it is better to use informed or heuristic search category for our experiment. The two algorithms in this category that we will used are A* and RBFS.

Also for the last many years there is another algorithm that gains lots of popularity and presumably outperformed the most used algorithm of A*. It is named as Potential Fields. This algorithm is now widely used due to its simplicity and mathematical formulations. We will use this algorithm as our third algorithm for our experiment.

2.3 Anytime Algorithms

Anytime- Algorithms (AA) is the term first introduced by Dean in 1980's, and are the type of algorithms that provide intelligent systems with the capability to consume their execution time for better quality of results. In anytime algorithms the quality of the results improves gradually with the increase in execution time. The word anytime is used because unlike normal algorithms, we can stop these algorithms at anytime and expect them to return an output.

Anytime Algorithms are best for use in real time environments, as these environments have some serious time and memory constraints and dynamically changing environments. As many computational tasks in RTS gaming are too complicated to be completed at real time speeds, anytime algorithms can help by intelligently allocating the computational time resources in the most effective way. Anytime Algorithms have certain set of properties and in order to declare an algorithm anytime, that algorithm has to satisfy these properties [7, 8 and 15].

2.3.1 AA Properties:

Listed below are the anytime properties that an algorithm should have to be considered as anytime.

- Measurable quality- means that whenever we stop an anytime algorithm the quality of its results should be measurable in some way and can be defined exactly.
- Mono-tonicity- is that the quality of an anytime algorithm's results should increases with increase in time and quality of the input.
- Consistency- is that the quality of an anytime algorithm's results are connected with computational time it have and the quality of the input.
- Interrupt-ability- is that we can stop an anytime algorithm at any time and it should provides us with some answer.
- Preempt-ability- is that an anytime algorithm can be stopped and can also be restarted again with minimal overhead.

2.3.2 Making AI Algorithms, Anytime – A Possible Solution?

The normal AI algorithms take too much computational time that it is almost impossible to use them in real time environments. The time these algorithms take to complete their calculation is too long for the dynamically changing environments like RTS games; because by the time these algorithms return their results, the environment may have changed completely. So we need algorithms those can react as the environment changes and anytime algorithms provide us with the possible solution; because we can stop these algorithms at anytime depending upon the current environmental changes and expect them to return some results, which is not possible with normal AI algorithms. So in order to use our selected AI algorithms in RTS gaming environments we first have to make them anytime because then we can better understand and optimized their behavior under RTS gaming environments.

This concludes our theoretical research of the study. From the next chapter we will start our implementation phase. In the next three chapters, we will take care of our three selected AI algorithms assigning one chapter for each. In each chapter we first briefly discuss the algorithm then define our implementation of the algorithm followed by our implementation of making it anytime.

CHAPTER 3: A - STAR SEARCH (A*)

3.1 Basic Concepts about A*

A – Star is an AI search algorithm that performs a systematic search through the search space to find an optimal path from the root node to the goal node. It belongs to the informed or heuristic search category as discussed in the previous chapter. The search algorithms in this category have some problem specific information about the environment apart from the problem definition. These search algorithms use this problem specific information to evaluate between the two nodes and possibly select the better one. The function used for this evaluation is known as evaluation function $f(n)$ [6, 10].

3.1.1 Evaluation Function $f(n)$

Evaluation function $f(n)$, is the function which uses the problem specific information about the environment to determine the preference of one node over the other and the formula used is

$$f(n) = g(n) + h(n)$$

In this formula, $g(n)$ is the exact cost of reaching from the root node to the current node n, and heuristic function $h(n)$ is the estimated cost to reach from this current node n to the goal node. This estimation of the cost is determined using the problem specific information about the environment that the A* algorithm has. The evaluation function $f(n)$ then can be expressed as the estimated cost of the cheapest path through node n, and also called as the f-cost of the node n. The cost can be anything and can be expressed in any unit. In RTS games, the cost can be the distance between the two points; those are expressed as nodes in search space. The cheapest path found using the A* algorithm is both complete and optimal if the heuristic function $h(n)$ satisfies certain constraints [6].

First among those constraints is that the heuristic function $h(n)$ needs to be admissible for the A* algorithm to be considered as complete and optimal. The heuristic function $h(n)$ is said to be admissible if it never overestimates the actual cost of reaching the goal node. The admissible heuristic functions are naturally optimistic because they always consider lesser total cost of the optimal path, than it actually is. As $g(n)$ is the exact cost of reaching from the root node to the node n and heuristic function $h(n)$ is admissible, then the resulted evaluation function $f(n)$ also never overestimate the actual cost to reach the goal node [6, 9].

To prove that A* algorithm is optimal when using admissible heuristic, let us suppose sG as a suboptimal goal node and $g(sG)$ as the exact cost of reaching from the root node to the current suboptimal goal node sG. Also consider $h(sG)$ as zero because for every goal node the estimated cost is always zero and let $C*$ is the cost of the optimal path from root node to the real goal node then

$$f(sG) = g(sG) + h(sG) = g(sG) > C*$$

The cost of the optimal path from root node to the suboptimal goal node sG is larger then the cost of the optimal path from root node to the real goal node. Thus suboptimal goal node sG will never be taken in as a goal.

Also let suppose node n as a node on the optimal path and if heuristic function $h(n)$ is admissible, then we know

$$f(n) = g(n) + h(n) \leq C*$$

So now we can show that,

$$f(n) \leq C* < f(sG)$$

A* Algorithm will never selected sG as a goal node because its f-cost is greater than the cost of the real goal node rather A* algorithm will select node n.

Second among the constraints is that the heuristic function $h(n)$ needs to be consistent for A* algorithm to be considered as optimal and complete. Heuristic function $h(n)$ is said to be consistent if for every node n the estimated cost of reaching the goal node should not be greater than the estimated cost of reaching the goal node from its successor node $n`$ plus the step cost of reaching from node $n$ to $n`$.

$$h(n) \leq c(n, a, n`) + h(n`)$$

In the formula, $c(n, a, n`)$ is the step cost of reaching from the node $n$ to its successor node $n`$ by using the action a, and $h(n`)$ is the estimated cost to reach from node $n`$ to the goal node. These two costs should not be greater than $h(n)$ the estimated cost to reach from node n to the goal node. This is also called as triangle inequality, in which the length of the one side should not be greater than the lengths of the other two sides [6].

Apart from using the evaluation function $f(n)$, A* algorithm also uses two lists, Open list and Close list for the systematic search of the search space. Close list stores all those nodes that are already been selected by the A* algorithm to be checked as the goal node and the Open list stores all the successors of those nodes in the Close list. These nodes in the Open list are sorted in an increasing order of their f-costs. The node with the least f-cost is selected next by the A* algorithm. So evaluation function $f(n)$ is the main driving force that guides the A* search in the search space [6, 9, 10].

3.2 Our Implementation of A*

A node is a basic building block of any search space; so before explaining the implementation of our A* algorithm, listed below is the pseudo code of our implementation of a *Node class*.

```
public class Node {
 public point state;     /* stores Node's position in xy Cord */
 public double fCost;   /* stores function cost of the Node */
 public double gCost;   /* stores total cost from root to the Node */
 public double hCost;   /* stores heuristic cost from Node to goal */
 public Node parent;    /* stores parent of the Node */
}
```

Table 3.1: Pseudo code of the Node class

As discussed in the previous section, the A* algorithm uses two lists for the systematic search of the search space. This pseudo code of a *Queue class* shows our implementation to provide us with the basic functionality of these lists.

```
public class Queue {
 public bool IsEmpty();
 public int Length();
 public Node GetFirstNode();
 public Node RemoveFirstNode();
 public void InsertNode(Node n);
 public void InsertAllNode(Node[] n);
 public void Sort();
}
```

Table 3.2: Pseudo code of the Queue class

Listed below is the pseudo code of our heuristic function $h(n)$ which simply is the straight line distance between the two nodes. This heuristic function is both admissible and consistent which is necessary for an optimal and complete A* algorithm.

```
public double Heuristic(Node Succ, Node Goal) {
 return
     Math.Abs(Succ.state.x - Goal.state.x) + Math.Abs(Succ.state.y - Goal.state.y);
}
```

Table 3.3: Pseudo code of the heuristic function h (n)

Admissibility is that the heuristic function $h(n)$ never overestimates the actual cost of reaching the goal node. Straight line distance between the two nodes is admissible as we know that the shortest distance between the two nodes is the straight line so the straight line distance can never be an overestimation. So our heuristic function is admissible.

Consistency of our heuristic function can also be proved by a simple example. Let’s suppose that we have the root node at $(0,0)$ and goal node at $(10,10)$ and let $(1,0)$ be the successor of the root node then by using heuristic function we calculate

$$h(n) \leq c(n, a, n`) + h(n`)$$
$$20 \;\leq\; 01 + 19$$

It satisfies the consistency equation. So our heuristic function is also consistent.

Now towards the end listed below is the complete pseudo code of our implementation of the A* algorithm. The A* algorithm can be called with the root and the goal nodes as its parameters. The first thing that the algorithm checks is if the root node is also our goal node or not? If yes then that means that we have found our goal and the algorithm simply returns with the goal success; however if not then the algorithm adds this root node in the Close list and calls the *ExpandNode* function.

```
public Node [] ExpandNode ( Node Root ) {
 Node [ 8 ] succ ;
 foreach (  Node  s  in  succ ) {
  s . state = A neighbouring point of the Root ;
  s . parent = Root ;
  s . gCost = Root . gCost + step cost from Root to s ;
  s . fCost = s . gCost + Heuristic ( s , Goal )
 }
}
```

Table 3.4: Pseudo code of the ExpandNode function of the A algorithm*

ExpandNode function as shown in the pseudo code creates the successor nodes of the node it is called for. It also calculates their states and f-costs and returns them as a node array. A* algorithm then conducts series of checks before adding these successor nodes in the Open list.

A* algorithm first checks whether any of the successor nodes is in the Open list or not? If yes then that means that this successor node is also a successor of another node in the Close list. In this case A* algorithm checks whether the f-cost of the current successor node is smaller then that in the Open list? If yes then that means that A* algorithm finds a cheaper path to reach the successor node. The successor node in the Open list is updated with the new f-cost. The same process is also carried out by checking all the successor nodes in the Close list. A successor node is only added in the Open list when it is neither in the Open nor in the Close list before. This whole process is necessary to make sure that A* algorithm didn’t search one node more than once and to stop node redundancy in the lists.

After these series of checks the Open list is sorted by increasing f-costs of the nodes in it. The node with the least cost is then selected to be checked as a goal node by recalling the

A* algorithm with this new selected node and the goal node. This process continues until we find the goal node.

```
public voi d AStar ( Node Root , Node Goa  l ) {
 if ( Root == Goal ) {
  goalSucc = true ;
  return ;
 }
 else {
  closeLis t . Insert Node ( Roo t ) ;
  Node [] successors = ExpandN ode ( Root ) ;
  foreach  ( Node  s  in  succ essors ) {
   if ( s == closeLi st [ Item ] ) {
    is_in_close = true ;
    if  ( s . fCost <= closeList [ Item ] . fCost )
      then  update the item with the new fCost
   }
   else if  ( s == o penList [ Item ] ) {
    is_in_open = tru e ;
    if  ( s . fCost <= openLis t [ Item ] . fCost )
     then  update the item with the new fCost
   }
   else if  ( is_in_close == f alse && is_in_open == false )
    openLi st . Inser tNode ( s ) ;
  }
  openList . Sort ( ) ; / * sorts openList by increasing fCost * /
  Node bes  tNode = op enList .  Re moveFirstN ode ( ) ;
  while ( goalSucc == false )
   AStar ( bestNode , Goal ) ;
 }
}
```
Table 3.5: Pseudo code of the AStar function of the A algorithm*

3.2.1 Local Minima Problem

When an algorithm gets stuck behind an obstacle in an environment such that it has no alternative position to go at, we said that the algorithm is in local minima. It is not possible for the algorithm to come out of this as it has no alternative position to go at because all the surrounding positions have higher f-costs values than the current position. Figure below shows a local minima situation.

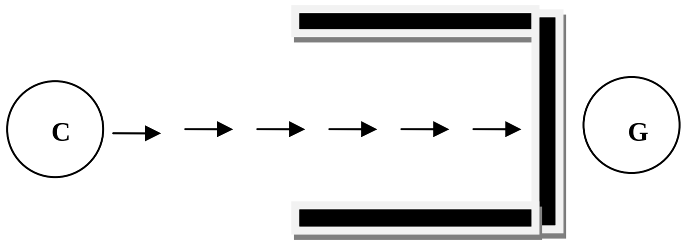


Figure 3.1: A local minima situation

The simplest way to come out of a local minima situation is to remember all the nodes the algorithm has already selected before. Algorithms those don't remember this cannot come out of the local minima because they keep on selecting the best node inside the local minima caring not that it has already been checked. A* algorithm however takes care of the already selected nodes by implementing the Close list. A* algorithm cannot select that node again which is in the Close list. Whenever A* algorithm stuck in the local minima it selects all the nodes (first those with least f-costs) in that local minima and puts them in the Close list. As A* algorithm cannot select them again, it fills the local minima up and eventually comes out of it as shown in the figure below.

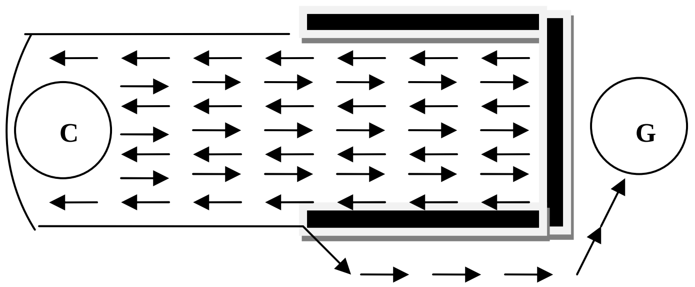


Figure 3.2: How A deals with the local minima situation*

In the above figure we can see that the A* algorithm not only fills up the local minima by selecting all the nodes in it but also selecting the nodes those are in between the local minima and the root node. This is because the A* algorithm is checking the f-costs of all the successor nodes in the Open list. Due to the local minima situation A* algorithm is forced to select a node whose f-cost is higher than the lowest f-cost in the Open list. As the Open list is sorted according to the lowest f-cost, this newly selected node with higher f-cost is placed below in the Open list and Open list has to select all the nodes above it before selecting that node again. This is one of the major drawback of the A* algorithm because whenever the A* algorithm stuck in a local minima situation it generates lots of nodes before coming out. Our implementation of the A* algorithm also behaves the same when stuck in a local minima situation.

3.3 Our Implementation of Anytime A*

The main structure of our implemented A* algorithm remains the same while converting it into anytime, because the basic working of the A* algorithm remains the same. The most important property of anytime algorithms is that, they can be stopped and then can be restarted at any time. So in order to make our implemented A* algorithm anytime we simply need to add a time constrain or a time limit to our algorithm. When the A* algorithm hits this limit it should be stopped and then restarted if desirable with new time limit. To avoid changes in our implemented A* algorithm, we implemented a sub-module class called as control manager class (*ctr_manager class*), which takes care of the time limit and the stopping and restarting of the A* algorithm. Listed below is our implementation of this class in the form of a pseudo code and our discussion about how it works.

As shown in the pseudo code, *ctr_manager class* implements a thread to add the time limit to the A* algorithm. The most important variable in this class is that of *new_Solution*, which stores the best solution found so far by the A* algorithm. The *ctr_manager class* can be called using the function *thread_AAStar,* which takes time limit, root node and the goal node as its parameters. If this function is called for the first time then the *new_Solution* is set equal to the root node, as root node is the best solution we know so far. The function *thread_AAStar* then starts the thread with the time limit, and the A* algorithm concurrently. The A* algorithm runs as long as this time limit allows. As soon as the time crosses the limit, the thread sets the *goalSucc* variable of the A* algorithm, which forces it to exit.

While the A* algorithm is running, it keeps on checking for more improved solutions than the one in the *new_Solution.* These improved solutions are judged by their distances from the goal node as shown in the pseudo code below.

As soon as the A* algorithm finds any improved solution it stores it in the *new_Solution* and keeps on looking for the more improved ones. When the time crosses the limit, the A* algorithm stops.

However now if we want to restart the A* algorithm, the *ctr_manager class* has to perform many checks before restarting the algorithm. Firstly the *ctr_manager class* has to check whether the A* algorithm finds an improved solution when running previously or not? If yes then it clears the Open and the Close lists, restarted the thread with the new time limit, and then calls the A* algorithm with this newly found improved solution as the root node. The *ctr_manager class* clears the Open and the Close lists because they contain nodes those are far away from the goal node then the currently selected node in the *new_Solution*, and A* algorithm didn't need to select them. However if the *ctr_manager class* finds out that the A* algorithm didn't find an improved solution when running previously then that means that the algorithm needs more time, so *ctr_manager class* didn't clear the Open and the Close lists.

```
public cla  ss ctr_man  ager {
 bool is_F  irst   =  true ;
 Node new_  Solution    =  null ;

 public vo  id run_ast  ar() {
   thread.  sleep ( sl  eep_Time )    ;
   goalSuc  c  =  true ;
 }
 public vo  id thread_  AAStar (   int  sleep_Tim  e , Node r  N , Node g  N ) {
  thread_a     =  new threa  d () ;
  if (is_F  irst) {
   new_Sol  ution    = rN;
   thread.  Start (run   _astar) ;
   AStar (   rN , gN )  ;
  }
  else if   ( rN    ==   new_Solut  ion) {
   //  keep  all the states  intact  because  algorithm   needs  more  time
   // to  find  a better  solution
   thread.  Start (run   _astar) ;
   AStar (   rN , gN);
  }
  else if   (rN !  =  new_Solut  ion) {
   //  algorithm   finds  an improve  solution,  clears  all the  previous  states
   // to  allow  algorithm   a fresh  start
   closeLi  st. Re moveAll ()    ;
   openLis  t. Re moveAll ()    ;
   thread.  Start (run   _astar);
   AStar (  new_Soluti   on , gN) ;
  }
 }
}
```
Table 3.6: Pseudo code of the ctr_manager class of the anytime A algorithm*

With this implementation we can stop and then can restart the A* algorithm, however anytime algorithm is not all about stopping and restarting an algorithm. To declare it anytime it should hold the properties of the anytime algorithms as discussed in the previous chapter. Before going into these properties firstly consider a sample test that we have conducted to check the anytime behavior of our implemented anytime A* algorithm. The results from this sample test is then used later on to prove the anytime nature of our algorithm.

Consider that our implemented anytime A* algorithm has to find the path between the root state $(10,10)$ and the goal state $(130,130)$. We will stop the algorithm after every $25m\sec$. The results are listed below:

Initial	After $25m\sec$	After $50m\sec$	After $75m\sec$
$(10,10)$	$(55,55)$	$(113,85)$	$(130,130)$
Distance from Goal d:- $(130,130)$	Distance from Goal d:- $(75,75)$	Distance from Goal d:- $(17,45)$	Distance from Goal d:- $(0,0)$

Table 3.7: Sample test 1 for the anytime A algorithm*

Now we will take anytime algorithm's properties one by one and see did our implemented anytime A* algorithm holds them or not?

- Measurable quality: This property means that the quality of the solution that an anytime algorithm returns, should be measure-able and represent-able in some way. In our above implemented anytime A* algorithm we measure this quality of the solution in terms of its straight line distance from the goal node. Lesser the distance of the solution from the goal node, better the solution is. Also this thing is shown by our sample test above. As long as we are providing more computational time to our implemented anytime A* algorithm it finds more improved solutions, the quality of those is determined by their reduction of the distance from the goal node.
- Mono-tonicity: This property means that the quality of the solution that an anytime algorithm returns, should increases with increase in computational time and quality of the input. This property can also be proved from the same results of the sample test as shown above. With the increase in computational time the quality of the solutions returned by our implemented anytime A* algorithm also improves.
- Consistency: According to this property the quality of the solution of an anytime algorithm is connected with computational time it have and the quality of the input. In other words we can say that if we run an anytime algorithm twice, each time providing with different computational time, the quality of the solution is better when run with longer computational time. To prove this we run our implemented anytime A* algorithm two times, each time providing with different computational time, the results are listed below

Initial	After $25m\sec$	After $50m\sec$	After $75m\sec$
$(10,10)$	$(51,53)$	$(93,122)$	$(130,130)$
Initial	After $50m\sec$	After $100m\sec$	After $150m\sec$
$(10,10)$	$(51,55)$	$(106,130)$	$(130,130)$

Table 3.8: Sample test 2 for the anytime A algorithm*

The quality of the solution is better when run with longer computational time which indicates that the quality of the solution is connected with the computational time our implemented anytime A* algorithm has.

- Interrupt-ability: This property means that for an algorithm to be declared as anytime we should be able to stop it at any time and it should provides us with some solution. Our implemented anytime A* algorithm is stoppable at any time and it will return a solution whenever it stops.

- Preempt-ability: According to this property an anytime algorithm can be stopped and can also be restarted again with minimal overhead. Our implemented anytime A* algorithm can also be restarted again once stopped. However it is difficult to calculate the overhead it have when restarted again.

Here, it is necessary to mention that we are only considering those properties of anytime algorithms that are most relevant to our study. Anytime algorithms, is a big field in itself and there is a long list of the properties that they can contain. Some of these properties are system specific and only related to those anytime algorithms specifically design for these systems. Others are more general and common properties and we are only considering these.

Now towards the very end we will discuss how our implemented anytime A* algorithm behaves when encounter a local minima situation. Unlike our implemented A* algorithm, the node generation of our implemented anytime A* algorithm is much less when stuck in the local minima. The behavior of our implemented anytime A* algorithm can be seen in the figures below

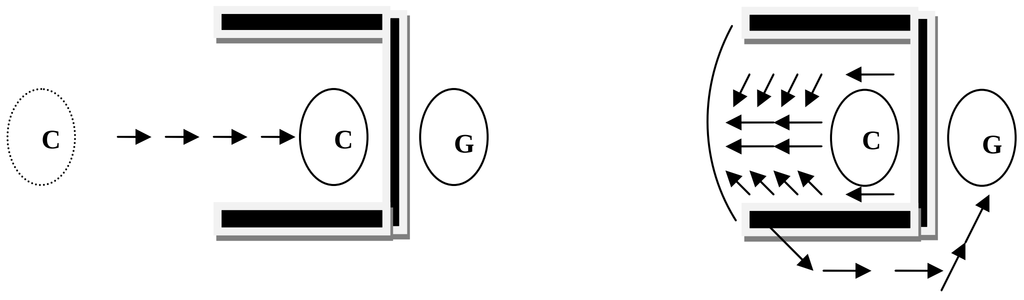


Figure 3.3: How anytime A algorithm deals with the local minima situation*

Unlike our implemented A* algorithm which generates nodes all the way back to the root node, our implemented anytime A* algorithm only generates that many nodes that allows it to fills up the local minima. This reduces the node generation a great deal when compares with implemented A* algorithm. This change in behavior is because of the change in the root node When implemented anytime A* algorithm is restarted after the first stop it updates its root node with the node which is closest to the goal node as shown above, and it also clears the old Open and the Close lists, that’s why it didn’t need to select all the nodes back to the initial root node.

CHAPTER 4: RECURSIVE BEST FIRST SEARCH (RBFS)

4.1 Basic Concepts about RBFS

RBFS is an AI algorithm that belongs to the informed or heuristic search category as discussed earlier in chapter 2. Like all the algorithms in this category, RBFS also uses the problem specific information about the environment to determine the preference of one node over the other. Like A* algorithm, RBFS algorithm also uses an evaluation function $f(n)$, which is calculated by the same formula of

$$f(n) = g(n) + h(n)$$

In this formula, $g(n)$ is the cost of reaching from the root node to the current node n, and heuristic function $h(n)$ is the estimated cost to reach from this current node n to the goal node which is calculated by using the problem specific information of the environment. The evaluation function $f(n)$ then is the estimated cost of the cheapest path through node n and called as the f-cost of the node n. Like A* algorithm, RBFS algorithm is also optimal and complete if the heuristic function $h(n)$ is admissible and consistent [6, 9].

The RBFS algorithm performs a systematic search in the search space to find the goal node but the way it performs this search is quite different than the A* algorithm. The RBFS algorithm is called using the root and the goal nodes as its parameters. But there is also a third parameter called as the limit, which stores an upper limit on the f-costs of the selected nodes. The RBFS algorithm cannot select any node with the higher f-cost than this limit.

First of all, the RBFS algorithm finds the successors of the root node, and if the least f-cost among these successor nodes is less than the limit then these successor nodes are stored in the Open list by increasing f-costs, and the root node is stored in the Close list. So the Close list stores all the selected nodes and the Open list stores all the successor nodes of the nodes in the Close list. Now the successor node with the least f-cost is taken as a best node and the successor node with second least f-cost is taken as an alternative node. The RBFS algorithm is recalled, with the best and the goal nodes, and also with the alternative node's f-cost, which now acts as the upper limit. This process continues until we find the goal node.

However if at any stage the least f-cost among the successor nodes of the best node is greater than the limit (alternative node's f-cost), then that means following this path is of no use as it will give us f-cost greater than what we can get by following the alternative node's path. In such case the RBFS algorithm updates the best node's f-cost with the least f-cost of its successor nodes and back track to use the alternative node's path. This update of the best node's f-cost with the least f-cost of its successor nodes is important because this will allow the RBFS algorithm to correctly asses the f-cost of this path if the path is reselected any time later in the algorithm [6, 10].

Unlike the A* algorithm the RBFS algorithm stores only nodes, those are in the current best node's path. Storing only the current path's nodes has some advantages and also some disadvantages. Due to this the RBFS algorithm is more memory efficient then the A* algorithm which has a major memory issue. However, the RBFS algorithm can't have any checks on the redundant nodes because it never stores all the nodes in the memory, due to this the RBFS algorithm can select the same node many times. And also if the RBFS algorithm has to backtrack many times during its search it also affects its performance [6, 9, and 10].

4.2 Our Implementation of RBFS

The Node in the RBFS search space has the same structure as that of in the A* algorithm. The pseudo code of the *Node class* has already been provided in the previous chapter. Also the Open and the Close list functionality is same as that of the A* algorithm, so its pseudo code (*Queue class*) has also been provided in the previous chapter. Also the heuristic function for the RBFS algorithm is exactly the same as that of the A* algorithm and its admissibility and consistency has already been proven in the previous chapter along side its pseudo code.

Now then, listed below is the pseudo code of the RBFS algorithm that we have implemented. The RBFS algorithm is called with the root and the goal node as its parameters plus with the limit parameter which provides us with an upper limit on the f-costs of the nodes in the selected path. In the RBFS algorithm, the root node is first compared with the goal node and if root node matches with the goal node then the algorithm returns with the goal success. However if no match found then the root node is expanded using the *ExpandNode* function. The basic functionality of the *ExpandNode* function alongside its pseudo code is explained well in detail in the previous chapter. The *ExpandNode* function returns the successor nodes of the root node with all the calculations of their f-costs.

Then we check whether the least f-cost of these successor nodes is less then the limit or not? If yes, then these successor nodes are added in the Open list and sorted with their increasing f-costs. Also the root node is added in the Close list. The successor node with the least f-cost is then taken as a best node and the successor node with the second least f-cost is taken as an alternative node. We recall the RBFS algorithm with the best and the goal node plus the alternative node's f-cost as a limit, and this process continues recursively until we find the goal node.

However if the least f-cost of the successor nodes is greater then the limit, then that means that we already know a path with lower f-cost then this one, thus the RBFS algorithm backtracks and updating the node with the least f-cost of its successors.

```
public double RBFS (Node Root , Node Goal , double limit) {
 if ( Root == Goal ) {
  goalSucc = true ;
  return Root . fCost ;
 }
 else {
  Node [] successors = ExpandNode ( Root ) ;
  successors . SortNodes ( ) ; /*sorts successors nodes by increasing fCost*/

  if ( successors [ firstNode ] . fCost > limit )
   return successors [ firstNode ] . fCost ;
  else {
   closeList . Insert ( Root ) ;

   foreach  ( Node  s  in  successors ) {
    if ( s  ! =  closeList [ item ] )
     openList . Insert ( s ) ;
   }
   openList . Sort ( ) ;
   Node bestNode = openList . RemoveFirstNode ( ) ;
   Node alternativeNode = openList . RemoveFirstNode ( ) ;

   while ( goalSucc == false ) {
    bestNode = RBFS ( bestNode , Goal , Math.Min ( limit , alternativeNode . fCost ) ) ;
    openList . Insert ( bestNode ) ;
    list . Sort ( ) ;
    bestNode = openList . RemoveFirstNode ( ) ;
    alternativeNode = openList . RemoveFirstNode ( ) ;
   }
  }
 }
}
```

Table 4.1: Pseudo code of the RBFS function of the RBFS algorithm

4.2.1 Local Minima Problem

Like A* algorithm, to avoid getting stuck behind an obstacle in an environment, we implemented the similar Close list in the RBFS algorithm, which stores all the nodes that are in the current path and are already been selected before. Whenever the RBFS algorithm selected a new node it first checks it in the Close list whether it has already been selected earlier or not? If yes then that selection is ignored and the RBFS algorithm selects the alternative node instead. With this method the RBFS algorithm can't select the same node again, which is necessary to come out of the local minima situation. Like the A* algorithm, the RBFS algorithm fills up the local minima by selecting all the nodes in it and eventually comes out of it as shown in the figure below

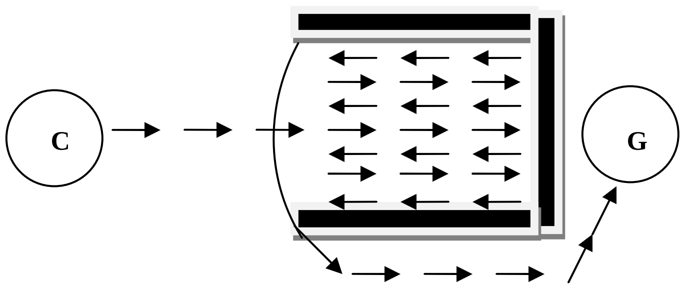


Figure 4.1: How RBFS algorithm deals with the local minima situation

Unlike the A* algorithm, the RBFS algorithm generates only that many nodes which are necessary to come out of the local minima. This is because in the RBFS algorithm node generation is controlled by the limit factor and the RBFS algorithm cannot select any node with f-cost greater than this limit. When ever the RBFS algorithm hits the local minima it increases the limit slightly just to produce few nodes with higher f-costs. If the RBFS algorithm finds the best node among those then it selects it, other wise the RBFS algorithm again increases the limit slightly and the process continues until the RBFS algorithm comes out of the local minima.

4.3 Our Implementation of Anytime RBFS

The process of making the RBFS algorithm anytime, is pretty much the same as that of the A* algorithm. To avoid many changes to our already implemented RBFS algorithm we uses the similar control manager class (*ctr_manager class*) as that for the A* algorithm. This *ctr_manager class* not only adds the time limit to the RBFS algorithm but also controls the stopping and the restarting of the RBFS algorithm.

We will not go into the real details of the *ctr_manager class* because it has already been discussed well in the previous chapter. The most important variable in this *ctr_manager class* is that of *new_Solution* which holds the best solution found so far by the RBFS algorithm. The *ctr_manager class* can be called using the *thread_ARBFS* function which takes time limit, root node, goal node and the upper limit as its parameters. The *thread_ARBFS* function then starts the thread with the time limit and the RBFS algorithm with root node, goal node and the upper limit as its parameter. When the RBFS algorithm starts, it keeps on checking for more improved solutions and placed it in the *new_Solution.* When the time limit is up the thread will set the *goalSucc* variable of the RBFS algorithm and the algorithm stops.

However if now we want to restart the algorithm the *ctr_manager class* will check whether the RBFS algorithm finds an improved solution when run previously or not? If yes then it will clear the Open and the Close lists and restart the thread with the new time limit, and also restart the RBFS algorithm with new improved solution as the root node. However if the RBFS algorithm didn't find any improved solution previously then *ctr_manager class* will not clear the Open and the Close lists, allowing the RBFS algorithm with more time to find the improved solution using these lists.

```
public class ctr_manager {
 bool is_First = true ;
 Node new_Solution = null ;

 public void run_rbfs() {
   thread .sleep ( sleep_Time) ;
   goalSucc = true ;
 }
 public void thread_ARBFS ( int sleep_Time , Node rN , Node gN , double limit) {
  thread_r = new thread () ;
  if (is_First) {
   new_Solution = rN;
   thread .Start (run_rbfs) ;
   RBFS (rN , gN , limit);
  }
  else if ( rN == new_Solution) {
   // keep all the states intact because algorithm needs more time
   // to find a better solution
   thread .Start (run_rbfs) ;
   RBFS (rN , gN , limit);
  }
  else if  (rN != new_Solution) {
   // algorithm finds an improve solution, clears all the previous states
   // to allow algorithm a fresh start
   closeList.RemoveAll ();
   openList.RemoveAll ();
   thread .Start (run_rbfs);
   RBFS (new_Solution , gN , limit) ;
  }
 }
}
```

Table 4.2: Pseudo code of the ctr_manager class of the anytime RBFS algorithm

The type of implementation of the anytime RBFS algorithm is same as that of the anytime A* algorithm. As anytime A* algorithm fulfills the anytime algorithm's properties, so with the same implementation anytime RBFS algorithm will also fulfill these properties. We are not discussing these properties again for the implemented anytime RBFS algorithm. Also our implemented anytime RBFS algorithm deals with the local minima situation similarly as that of our implemented RBFS algorithm

CHAPTER 5: POTENTIAL FIELDS (PF)

5.1 Basic Concepts about PF

Potential fields, is a behavior based algorithm that always looks at the environment for changes and reacts with it. There are two types of behavior based algorithms, Always- On and Sometimes- On algorithms [13].

- Always- On algorithms represent behaviors those always observe the environment for changes and act accordingly.
- Sometimes- On algorithms, however represent behaviors those only activated when some environmental change triggers them, and then act accordingly.

So potential fields is a behavior based always- on algorithm that constantly looks at its environment for changes. To elaborate this with an example, consider potential field as a ball rolling down a hill, an environment. The movement of the ball depends upon the geometry of the hill. As the geometry changes, so does the movement of the ball. Thus the behavior exhibit by the ball depends on its environment.
But unlike this behavior of the ball, the behavior of a gaming character is determined by a game designer and created in three main steps [13].

- Create multiple behaviors each assigning a particular task
- Represent a potential field for each of the above created behaviors.
- Combine these potential fields to produce complex gaming character's moves.

Now with the help of simple examples we will see how we can represent behaviors as potential fields and how we can combine these potential fields to produce complex moves.

5.1.1 Representing Behaviors as Potential Fields

The first step among the three step process is to create multiple behaviors, those we want to represent as potential fields. Let's suppose we have created two behaviors, seekGoal and avoidObstacle and now we want to represent them as potential fields.

The main part of a potential field is an action vector. As we know a vector has two parts, its magnitude and its direction. In an action vector the magnitude represents the speed of a gaming character moving towards the goal and direction represents the direction in which it should move. Each created behavior outputs an action vector at each point in the environment, and the collection of these action vectors constitutes a potential field for that behavior. Now let's suppose for the seekGoal behavior we have an action vector $= [\Delta x, \Delta y]$ .Its magnitude and direction can be calculated as follows [13]

- Let (x, y) be the position of the gaming character.
- Let (xG, yG) be the position of the goal and r be the radius of that goal. Let s be the spread of the potential field and α be the strength of the potential field.
- First find the distance between the character and the goal by, $d = \sqrt{(xG - x)^2 + (yG - y)^2}$

- Then find the angle between the character and the goal by, $a = \tan^{-1}\left(\frac{yG - y}{xG - x}\right)$
- And then at the end set the action vector $(\Delta x, \Delta y)$ according to the following,

  If $d < r$ $\qquad \Delta x = \Delta y = 0$

  If $r \le d \le s + r$ $\qquad \Delta x = \alpha(d - r)\cos(a)$ and $\Delta y = \alpha(d - r)\sin(a)$

  If $d > s + r$ $\qquad \Delta x = \alpha\, s\cos(a)$ and $\Delta y = \alpha\, s\sin(a)$

If we find action vector for the seekGoal behavior at every point in the environment then the resulting potential field will look something like in the figure below. Here each arrow represents an action vector for the seekGoal behavior at that particular point of the environment.

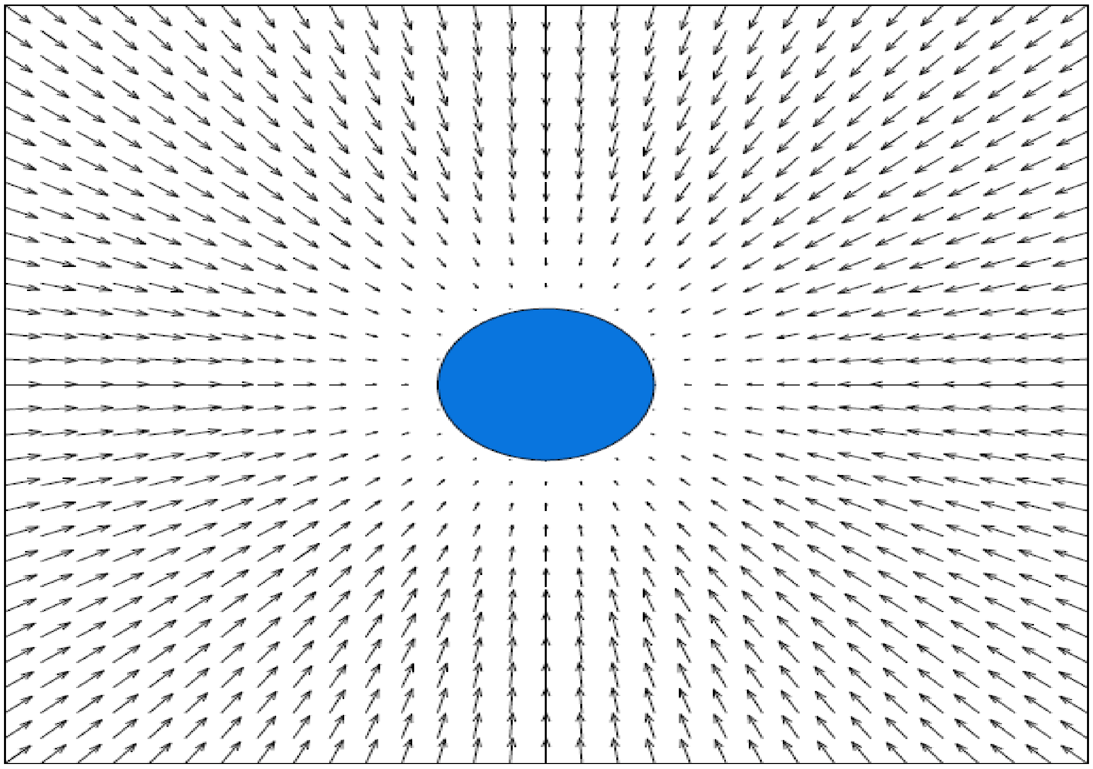


Figure 5.1: The resulting potential field for the seekGoal behavior [13].

Now let us consider our second created behavior of avoidObstacle. We can represent this behavior in potential field using the following steps

- Let (x, y) be the position of the gaming character.
- Let (xO, yO) be the position of the obstacle, r be the radius of that obstacle Let s be the spread of the potential field and β be the strength of the potential field.
- First find the distance between the character and the obstacle by, $d = \sqrt{(xO - x)^2 + (yO - y)^2}$
- Then find the angle between the character and the obstacle by, $a = \tan^{-1}\left(\frac{yO - y}{xO - x}\right)$

- And then at the end set the action vector $(\Delta x, \Delta y)$ according to the following,
  If $d < r$ $\quad \Delta x = -sign(\cos(a))\infty$ and $\Delta y = -sign(\sin(a))\infty$
  If $r \le d \le s + r$ $\quad \Delta x = -\beta(s + r - d)\cos(a)$ and $\Delta y = -\beta(s + r - d)\sin(a)$
  If $d > s + r$ $\quad \Delta x = \Delta y = 0$

The resulting potential field for the avoidObstacle behavior looks something like in the figure below

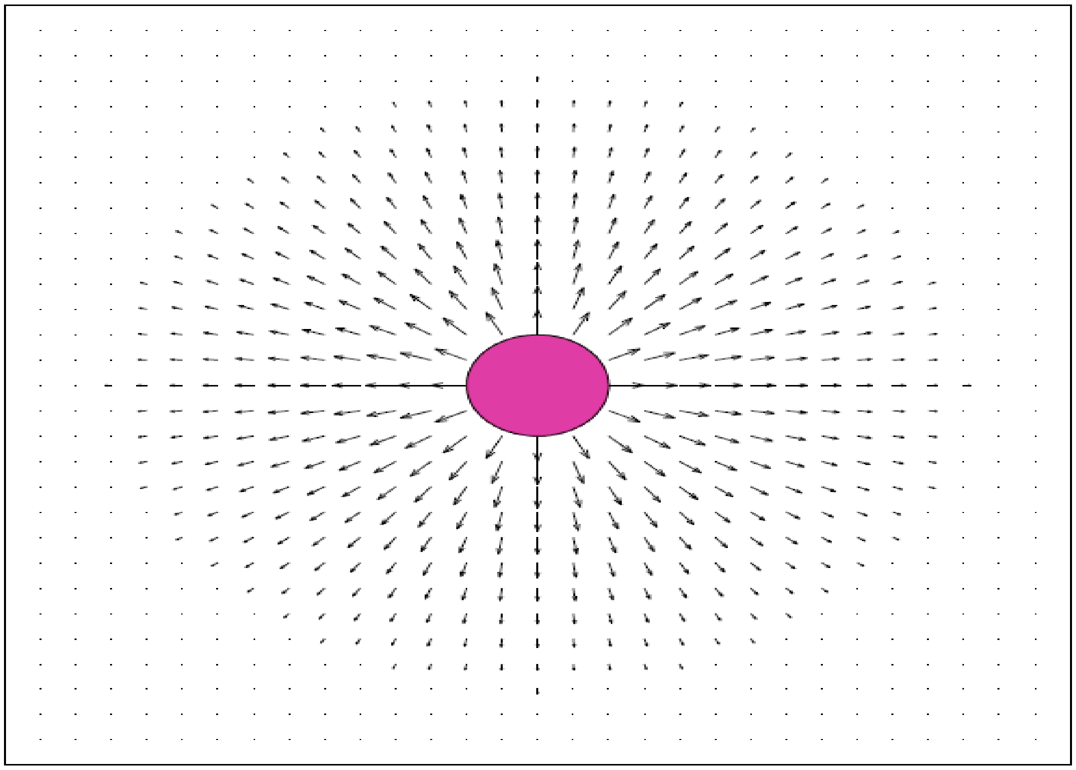


Figure 5.2: The resulting potential field for the avoidObstacle behavior [13]

5.1.2 Combining Potential Fields

So far we have considered two behaviors, seekGoal and avoidObstacle and created a potential field for each of them. As most of the games have both these behaviors in their environments, we have to find a way to combine their potential fields so that our character can avoid obstacles and find goal at the same time. As we discussed earlier potential fields are not more than a collection of action vectors. So combining two potential fields can be easily achieved by combining action vectors for both the fields [13].

Let's consider $(\Delta_G x, \Delta_G y)$ be the potential field generated by the seekGoal behavior and $(\Delta_O x, \Delta_O y)$ be the potential field generated by the avoidObstacle behavior, then by combining these two we get,

$$\Delta x = \Delta_G x + \Delta_O x$$
$$\Delta y = \Delta_G y + \Delta_O y$$

The resulted combined potential field and possible path taken by a character towards its goal is shown in the figure below

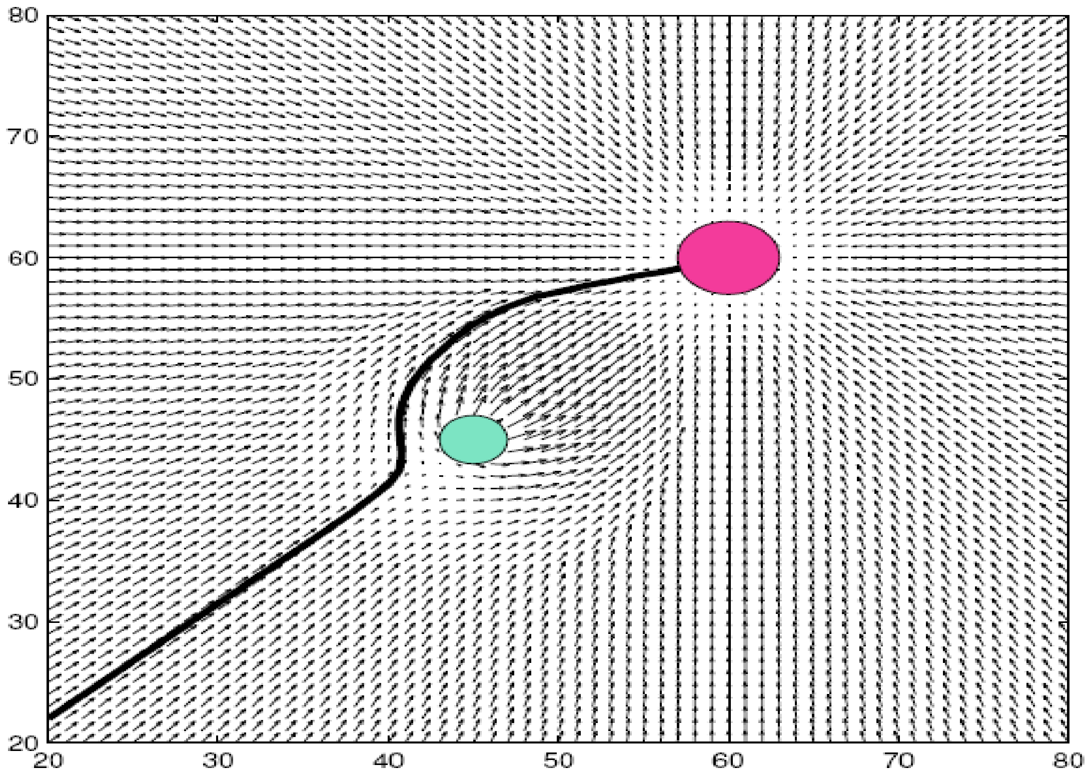


Figure 5.3: Combined potential fields & possible character path to goal [13]

5.2 Types of Potential Fields

Up till now, we have seen two types of potential fields, an attractive potential field for the seekGoal behavior and the repulsive potential field for the avoidObstacle behavior. However there exist many other types of potential fields; some of them are discussed below [13].

5.2.1 Uniform Potential Field

A very useful and simple type of potential field that can be used for various forms of behaviors like run away or follow a wall or return back to the base etc.

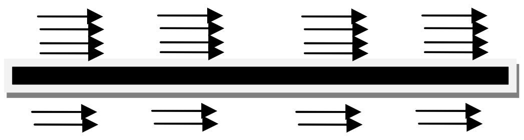


Figure 5.4: Uniform Potential Field

5.2.2 Perpendicular Potential Field

This type of potential field is useful to depict behavior such as to avoid walls or to get away from an enemy base etc.

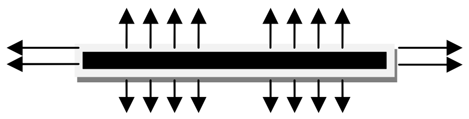


Figure 5.5: Perpendicular Potential Field

5.2.3 Tangential Potential Field

This type of potential field creates a circular field around an obstacle or a goal and useful to depict behaviors such as go around an obstacle or patrolling own base etc.

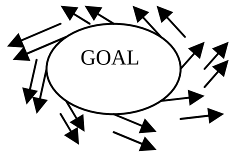


Figure 5.6: Tangential Potential Field

5.2.4 Random potential Field

This potential field is useful in avoiding local minima situation, or if you want character to wonder around in random directions.

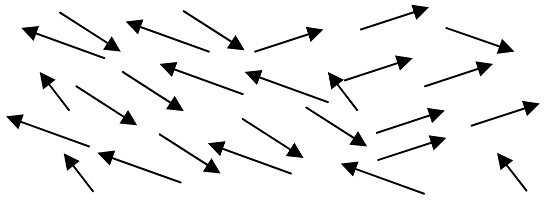


Figure 5.7: Random Potential Field

5.3 The Grid

So far we have considered gaming environments with a single goal and an obstacle. But in a RTS gaming environments, a character has to fulfill multiple goals and avoid multiple obstacles at the same time. A character has to make selection decisions between two possible goals. Also the preference of one goal over the other may also change as the environment changes. Each of these goals and obstacles in the environment has its own potential field. As the number of objects in a gaming environment increases so is the difficulty to manage all the potential fields together. In such cases the above mentioned implementation of potential fields didn’t work, and we need to handle it in a different way. One possible way to overcome this is to divide the whole gaming environment in a grid. Each square in this grid represent a point in the environment, and contains the combined potential fields for all the objects, on that point. There exists two ways to implement a grid [14].

In the first way of implementation, we calculate the grid of whole of the environment, and the way to do this is through this formula

$$PF(x, y) = \left(\frac{c}{d^2} \right)$$

Where c shows the importance of an object in a gaming environment; and d shows the distance between the character and that object. After calculating all the potential fields on one point we can combine them through this formula

$$P_{PF} = \sum PF(x, y)$$

Although this type of implementation may suitable for games with static environments, those didn’t change rapidly but for RTS gaming environments this way of implementation is not suitable

The second way to implement the grid is instead of calculating grid for the whole of the environment, we calculate only those points near the character. The formulas we can use are the same as written above. Although this way of implementing the grid consumes less computational power but it may also be difficult as we constantly have to take care of the obstacles in the environment [14].

5.4 Our Implementation of Potential Field

In the previous sections, we have discussed two slightly different ways of implementing the potential fields. The difference in implementation is largely due to the different kinds of gaming environments, a character can encounter. Therefore before going towards the implementation details it is necessary to fully understand the environment that we want our potential fields to run on. The environment that we will use has a single goal with multiple obstacles to avoid. We believe that the grid way of implementing the potential fields is more suitable for the environments having multiple goals and obstacles in it. Although you can modify your implementation of the grid for the single goal environment but still we are using the first way of implementing the potential fields because we believe that it is simpler than to implement the grid.

The two obvious behaviors that we have created for our implementation of potential fields are the seekGoal and the avoidObstacle. Discussed below is our implementation of potential fields for both these behaviors and how we have combined them to get complex character's moves.

5.4.1 Creating Potential Field for seekGoal Behavior

The steps we are using for creating potential field for the seekGoal behavior are the same as indicated above. First we find the distance d between the character (x, y) and the goal (xG, yG) by using the formula

$$d = \sqrt{(xG - x)^2 + (yG - y)^2}$$

And then find the angle a between the character (x, y) and the goal (xG, yG)
.

$$a = \tan^{-1}\left(\frac{yG - y}{xG - x}\right)$$

And towards the end we have calculated the action vector $(\Delta_G x, \Delta_G y)$ for any given point using the same three formulas

If $d < r$ $\quad \Delta_G x = \Delta_G y = 0$
If $r \le d \le s + r$ $\quad \Delta_G x = \alpha(d - r)\cos(a)$ and $\Delta_G y = \alpha(d - r)\sin(a)$
If $d > s + r$ $\quad \Delta_G x = \alpha\, s\cos(a)$ and $\Delta_G y = \alpha\, s\sin(a)$

Our implementation of these formulas for the seekGoal behavior is listed below in form of a pseudo code

```
public double getDistance( Node R , Node G ){
  return Math.Sqrt( Math.Pow(( G.x−R.x ), 2 )+Math.Pow (( G.y−R.y ), 2 ));
}

public double getAngle ( Node R , Node G ){
  return Math.Atan2 (( G.y − R.y ), ( G.x − R.x ));
}

public void seekGoal ( Node N ){
  N.d = getDistance ( N , GoalPoint );
  N.a = getAngle ( N , GoalPoint );
  if ( N.d < Gr ){
    $\Delta_G x = 0$ ;
    $\Delta_G y = 0$ ;
    GoalSucc = true ;
  }
  else if (( N.d ≥ Gr ) && ( N.d ≤ ( Gs + Gr ))){
    $\Delta_G x$ = ( N.d − Gr ) * Math.Cos ( N.a );
    $\Delta_G y$ = ( N.d − Gr ) * Math.Sin ( N.a );
  }
  else if ( N.d > ( Gr + Gs )){
    $\Delta_G x$ = Gs * Math.Cos ( N.a );
    $\Delta_G y$ = Gs * Math.Sin ( N.a );
  }
}
```

Table 5.1: Pseudo code of the seekGoal function of the PF algorithm

5.4.2 Creating Potential Field for avoidObstacle Behavior

The steps we have taken to implement potential fields for the avoidObstacle behavior are slightly different than what we have discussed earlier. The formulas for the avoidObstacle behavior we have discussed in the previous sections are more suitable for the obstacles those are round in shape and have a constant radius between the center and the boundaries of the obstacle. But in our environment the obstacles are more like rectangular in shape, so we have to adopt a different strategy to create potential fields for such kind of obstacles.

The steps that we have adopted for this purpose are better been illustrated with an example.

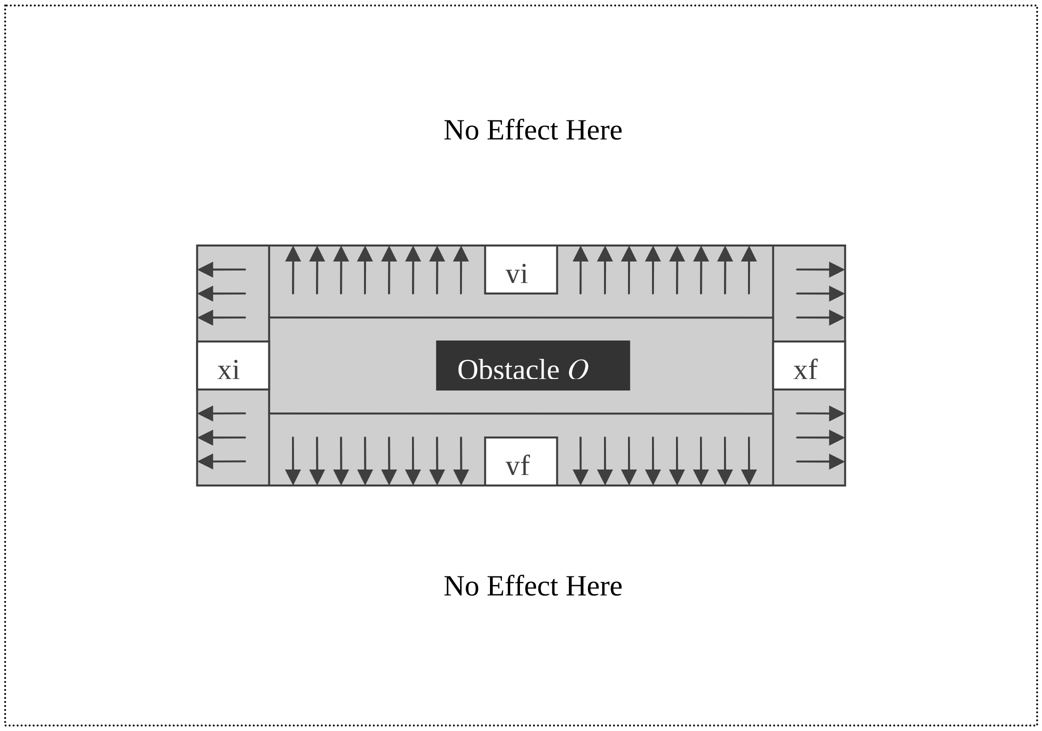


Figure 5.8: Potential Field generated by an obstacle in an environment.

Consider the rectangular shaped obstacle O situated in an environment. In order to create potential field for such kind of object we divided the region around the object into four parts *xi,xf,yi and yf* .Then created a perpendicular potential field away from the obstacle in each part. Outside this region the potential field of the obstacle has no effect. The width of this region is also adjustable. The formulas that we have used and the pseudo code for the implementation of this potential field are listed below.

Let (x, y) be the position of a character and let r be the width of the region around the obstacle. And let xi, xf, yi, yf be the four parts of the region and $(\Delta_O x, \Delta_O y)$ be our action vector for the avoidObstacle behavior then

$$\text{If } \left\{\begin{matrix} x < (xi - r) \,\|\, x > (xf + r) \,\| \\ y < (yi - r) \,\|\, y > (yf + r) \end{matrix}\right\} \qquad \Delta_O x = \Delta_O y = 0$$

$$\text{If } \left\{\begin{matrix} x \geq (xi - r)\,\&\& \\ x \leq (xi) \end{matrix}\right\} \qquad \Delta_O x = -(x - (xi - r)) \text{ and } \Delta_O y = 0$$

$$\text{If } \left\{\begin{matrix} x \leq (xf + r)\,\&\& \\ x \geq (xf) \end{matrix}\right\} \qquad \Delta_O x = +((xf + r) - x) \text{ and } \Delta_O y = 0$$

$$\text{If } \left\{\begin{matrix} y \geq (yi - r)\,\&\& \\ y \leq (yi) \end{matrix}\right\} \qquad \Delta_O x = 0 \text{ and } \Delta_O y = -(y - (yi - r))$$

$$\text{If } \left\{\begin{matrix} y \leq (yf + r)\,\&\& \\ y \geq (yf) \end{matrix}\right\} \qquad \Delta_O x = 0 \text{ and } \Delta_O y = ((yf + r) - y)$$

```
public voi d avoidObs tacle $(Node\ N)$ {
 if $(N.x < (xi - r) \| N.x > (xf + r) \| N.y < (yi - r) \| N.y > (yf + r))$ {
   $\Delta_O x = 0$ ;
   $\Delta_O y = 0$ ;
 }
 else if $((N.x \geq (xi - r))\ \&\&\ (N.x \leq (xi)))$ {
   $\Delta_O x = -(N.x - (xi - r))$;
 }
 else if $((N.x \leq (xf + r))\&\&(N.x \geq (xf)))$ {
   $\Delta_O x = ((xf + r) - N.x)$;
 }
 else if $((N.y \geq (yi - r))\&\&(N.y \leq (yi)))$ {
   $\Delta_O y = -(N.y - (yi - r))$;
 }
 else if $((N.y \leq (yf + r))\&\&(N.y \geq (yf)))$ {
   $\Delta_O y = ((yf + r) - N.y)$;
 }
}
```

Table 5.2: Pseudo code of the avoidObstacle function of the PF algorithm

After creating the potential fields of the goal and the obstacles in the environment we combine them together like this

$$\Delta x = \Delta_G x + \sum \Delta_O x$$
$$\Delta y = \Delta_G y + \sum \Delta_O y$$

5.4.3 Local Minima Problem

To overcome a local minima situation for the potential field we have used trees with the best first search technique. To achieve this we have created two lists, close and open list. In the close list we have stored all the previously selected nodes by our potential field and in the open list we have stored all the successors of those nodes.

When a node is been selected by our potential field it is first checked in the close list. If that node has not found in the close list then that means it is not been selected before. We insert that node in the close list and all its successors in the open list and recall the potential field for the generation of a new node. However if this node is found in the closed list then it means that this point has already been selected earlier by the potential field, then in this scenario we sort our open list and select the point with the lowest distance from the goal. Put this node in the closed list and recall the potential field with this new node.

Using this way, whenever our potential fields encounters a local minima situation it fill ups the local minima by selecting all the nodes in it. Eventually it comes out of the local minima and finds the goal. Now towards the end we provide the complete pseudo code of our potential field algorithm.

```
public void PFields (Node Root, Node Goal){
  openList = closeList = null;
  GoalSucc = false;
  if (Root == Goal){
    GoalSucc = true;
    return;
  }
  foreach (Node o in openList){
    if (o == Root): remove o from openList;
  }
  foreach (Node c in closeList){
    if (c == Root): is_in_cl = true;
    else: is_in_cl = false;
  }
  if (is_in_cl == false){
    closeList.Insert(Root);
    Node[] successors = Expand(Root);
    foreach (Node s in successors)
      openList.Insert(s);
    openList.Sort();
    seekGoal(Root);
    avoidObstacle(Root);
    Root.x += $\Delta_G x + \Delta_O x$;
    Root.y += $\Delta_G y + \Delta_O y$;
    if (GoalSucc == false)
      pFields(Root);
  }
  else if (is_in_cl == true){
    if (GoalSucc == false){
      bestNode = openList.RemoveFirst();
      PFields(bestNode);
    }
  }
 }
}
```

Table 5.3: Pseudo code of the PFields function of the PF algorithm

5.5 Our Implementation of Anytime PF

Similarly like anytime A* and anytime RBFS algorithms we have created a control manager class (*ctr_manager class*) for the potential field algorithm, that not only adds the time limit to the algorithm but also takes care of the stopping and the restarting of the algorithm.

```
public class ctr_manager {
 bool is_First = true ;
 Node new_Solution = null ;

 public void run_pfields() {
   thread. sleep ( sleep_Time ) ;
   goalSucc = true ;
 }
 public void thread_APFields ( int sleep_Time , Node rN , Node gN ) {
  thread_p = new thread () ;
  if (is_First) {
   new_Solution = rN;
   thread. Start (run_pfields) ;
   PFields ( rN , gN );
  }
  else if ( rN == new_Solution) {
   // keep all the states intact because algorithm needs more time
   // to find a better solution
   thread. Start (run_pfields) ;
   PFields ( rN , gN );
  }
  else if (rN ! = new_Solution) {
   // algorithm finds an improve solution, clears all the previous states
   // to allow algorithm a fresh start
   closeList. RemoveAll () ;
   openList. RemoveAll () ;
   thread. Start (run_pfields);
   PFields (new_Solution , gN) ;
  }
 }
}
```

Table 5.4: Pseudo code of the ctr_manager class of the anytime PF

The functionality of the *ctr_manager class* is discussed well in details in the previous two chapters so we are not discussing it again here for the potential field algorithm. The pseudo code of the *ctr_manager class* is listed above

CHAPTER 6: PFPC – OUR OWN BUILD PLATFORM

In this chapter we will spend sometime discussing our platform, that is specially been built to conduct our experiments. The chapter is purely been devoted explaining our platform, its features and ways to use it.

6.1 Path Finding Performance Comparison (PFPC) Platform

Path Finding Performance Comparison (PFPC) is a research platform specifically designed to study AI algorithms or anytime AI algorithms for the AI problem of path finding. PFPC is developed by using the language Visual C# and the tool Microsoft Visual C# Express Edition 2005 which is available free of cost from the Microsoft website **http://www.microsoft.com/express/2005/download/default.aspx**

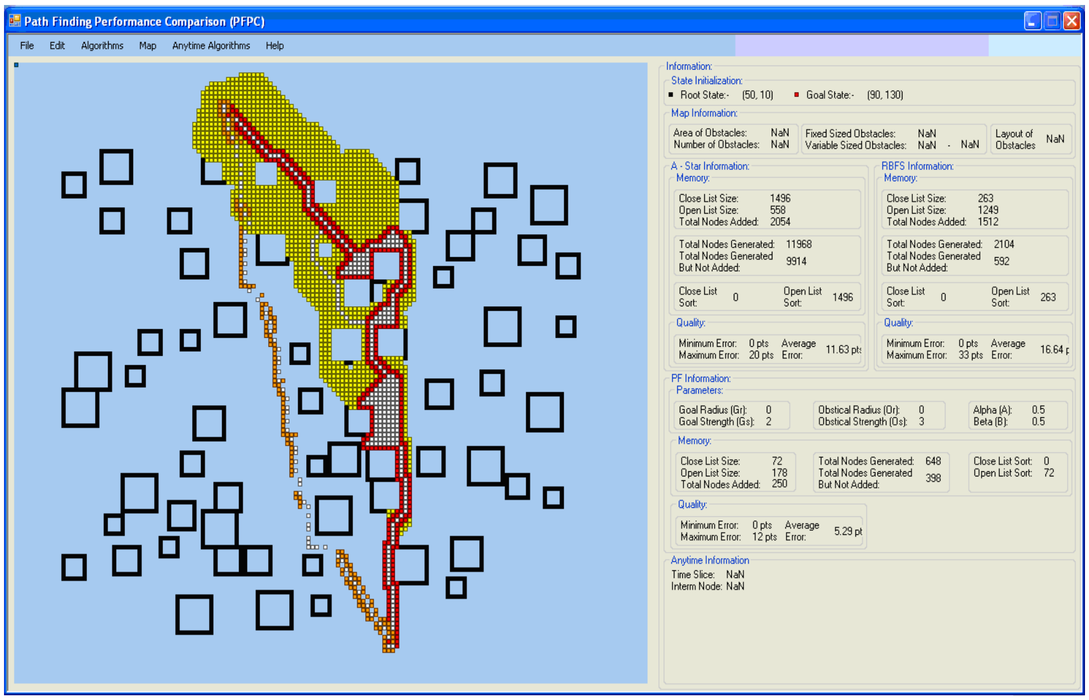
In this section we will take a brief look at some of the features of the PFPC platform that can be used to perform the performance comparisons of different AI algorithms or their anytime counterparts for the path finding AI problem. Shown below is the main window of this platform.

The main window of PFPC platform is divided into two main sub windows; on the left of the screen is the environment window and on the right is the information window.

Environment window provides us with the visual feedback of the AI algorithms in use. As shown below, the different AI algorithms move around in the environment and are represented with different colours. This window helps us a great deal to visually analyze the behaviour of these AI algorithms and how they find their way towards the goal node. The rectangular shaped objects are the obstacles that these AI algorithms need to avoid on their way to the goal node.

Information window on the other hand provides us with all the statistical data about the performances of these running AI algorithms. It also shows us the settings of the root and the goal nodes as well as the entire map related information that is been loaded into the environment.

A map in the PFPC platform consists upon obstacles with different dimensions, sizes and layouts. A PFPC user can generate and then load a new map into the environment window using the new map window. This window as shown below can be found using the main menu.



Information:

State Initialization:
■ Root State:- (50, 10) 🟥 Goal State:- (90, 130)

Map Information:
Area of Obstacles: NaN
Number of Obstacles: NaN
Fixed Sized Obstacles: NaN
Variable Sized Obstacles: NaN - NaN
Layout of Obstacles NaN

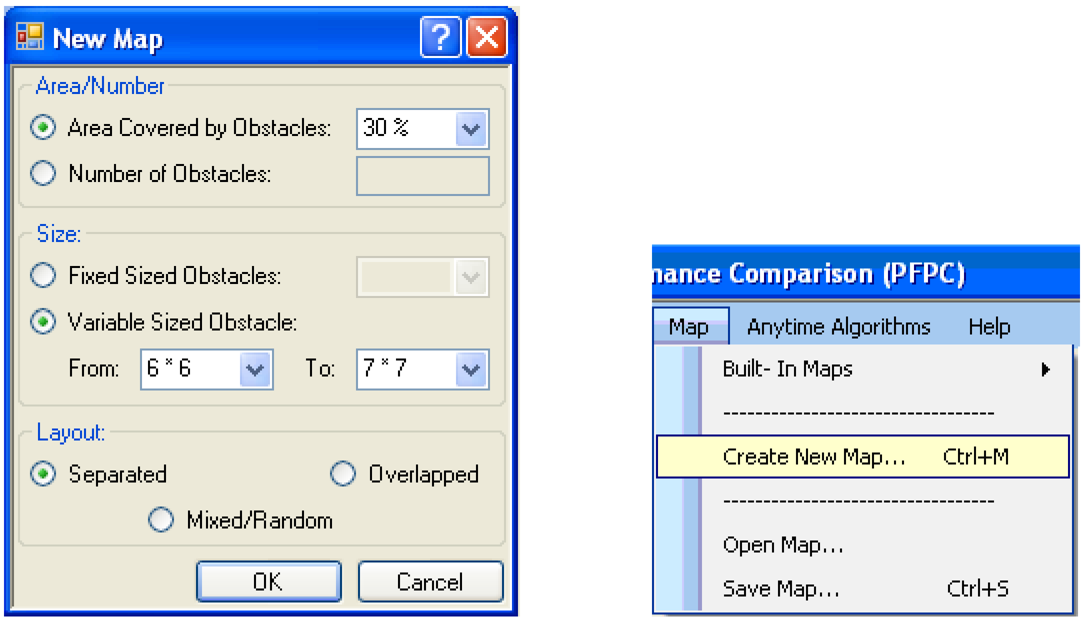
A - Star Information:
Memory:
Close List Size: 1496
Open List Size: 558
Total Nodes Added: 2054
Total Nodes Generated: 11968
Total Nodes Generated But Not Added: 9914
Close List Sort: 0 Open List Sort: 1496
Quality:
Minimum Error: 0 pts Maximum Error: 20 pts Average Error: 11.63 pts

RBFS Information:
Memory:
Close List Size: 263
Open List Size: 1249
Total Nodes Added: 1512
Total Nodes Generated: 2104
Total Nodes Generated But Not Added: 592
Close List Sort: 0 Open List Sort: 263
Quality:
Minimum Error: 0 pts Maximum Error: 33 pts Average Error: 16.64 p

PF Information:
Parameters:
Goal Radius (Gr): 0
Goal Strength (Gs): 2
Obstical Radius (Or): 0
Obstical Strength (Os): 3
Alpha (A): 0.5
Beta (B): 0.5
Memory:
Close List Size: 72
Open List Size: 178
Total Nodes Added: 250
Total Nodes Generated: 648
Total Nodes Generated But Not Added: 398
Close List Sort: 0
Open List Sort: 72
Quality:
Minimum Error: 0 pts Maximum Error: 12 pts Average Error: 5.29 pt

Anytime Information
Time Slice: NaN
Interm Node: NaN

Figure 6.1: The main window of the PFPC platform



Area/Number
[✓ Area Covered by Obstacles:] 30 %
Number of Obstacles:

Size:
Fixed Sized Obstacles:
[✓ Variable Sized Obstacle:]
From: 6 * 6 To: 7 * 7

Layout:
[✓ Separated] Overlapped
Mixed/Random

Figure 6.2: The new map window & the main menu button for that window

There are various options available for the PFPC user in the new map window and can be divided into three categories.

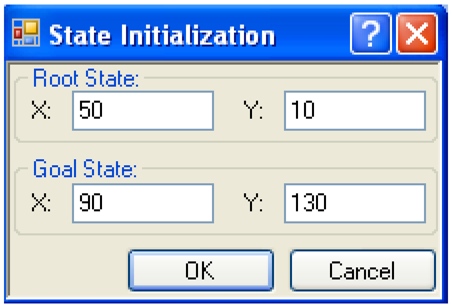
- Area / Number: In this category the PFPC user can select the total area covered by the obstacles in percentage, in the environment or it can specify the total number of obstacles it wants in the environment.
- Size: In this category the PFPC user can select the sizes of the obstacles in the environment. The obstacles can be of fixed or variable sizes. By selecting the

fixed sized obstacles, all the obstacles in the environment are of the same size. However if the user selects the variable sized obstacles then the obstacles are of different sizes ranging between the provided inputs of the user.

- Layout: In this category the PFPC user can select the layout of the obstacles in the environment. There are three options available in this category, the user can select between the separated, overlapped or mixed/random layouts. Selecting separated means that all the obstacles in the environment will be separated from each other. Selecting overlapped means that all the obstacles in the environment will be overlapped on each other and by selecting mixed/random as a layout, it means that the obstacles will be placed randomly in the environment.

Besides creating new maps, the PFPC user can also save and then reopen them if wanted for the future use. There are also available some built in maps for the user that can be used. Apart from all this the user can create a new map directly on to the environment window simply by left clicking and dragging the mouse over it.

After loading the desired map into the environment the PFPC user can now initialize the root and the goal nodes for the AI algorithms, using the state initialization window. This window as shown below can be found using the main menu.


State Initialization

Root State:
X: 50 Y: 10

Goal State:
X: 90 Y: 130

OK Cancel

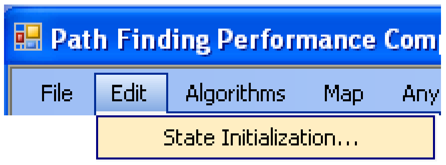
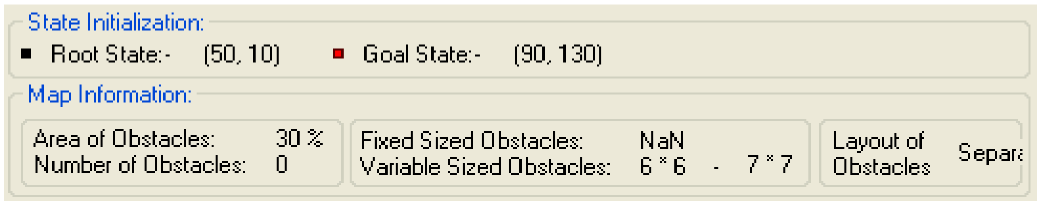


Figure 6.3: The state initialization window & the main menu button for that window

While the PFPC user is performing all these actions the information window is also updating side by side as shown below


State Initialization:

- Root State:- (50, 10)
- Goal State:- (90, 130)

Map Information:

Area of Obstacles: 30 %
Number of Obstacles: 0

Fixed Sized Obstacles: NaN
Variable Sized Obstacles: 6 * 6 - 7 * 7

Layout of Obstacles Separa

Figure 6.4: The information window showing the updated state and map information

Now after loading the map into the environment and initializing the root and the goal nodes, the PFPC user can select any of the AI algorithms implemented in the platform. The three AI algorithms implemented so far are the A*, RBFS and the Potential Fields. The user can select any one among these as shown below using the main menu

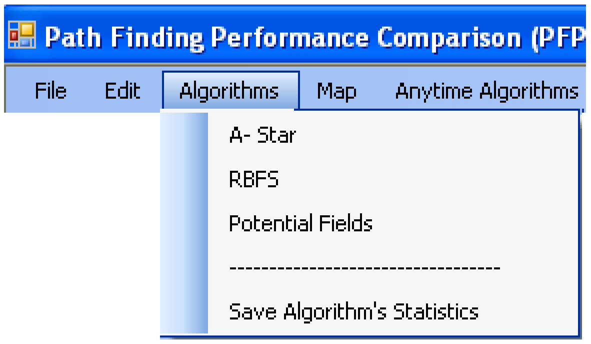
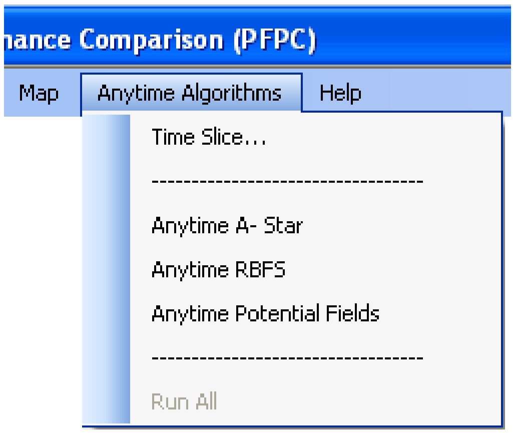

Path Finding Performance Comparison (PFP
File
Edit
Algorithms
Map
Anytime Algorithms
A- Star
RBFS
Potential Fields
Save Algorithm's Statistics

Figure 6.5: The main menu buttons for the AI algorithms selection

The user can also select any of the implemented anytime AI algorithms to run on the selected environment. The anytime algorithms can be found using the main menu



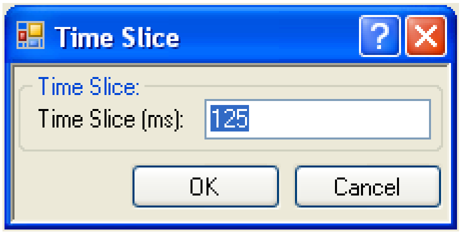
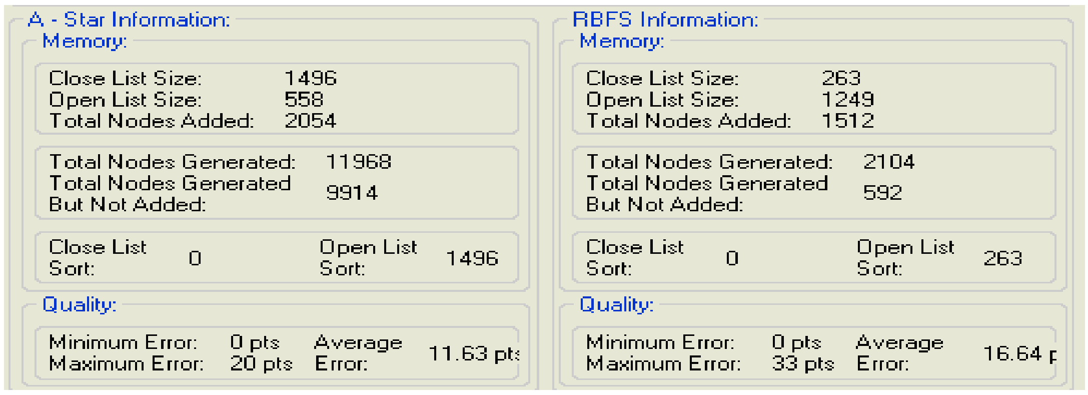


Figure 6.6: The time slice window & the main menu window for the anytime algorithm selection

But before selecting any of the implemented anytime algorithms the user has to specify the time limit it wants the anytime AI algorithm to run for. Figure above shows a time slice window used for this purpose. The time added in the window is in milliseconds.

After selecting any of the implemented AI algorithms or anytime AI algorithms, the algorithm will run on the environment and tries to find the goal node. The information window on the right side of the main window is also then updated showing different statistics of the algorithm.



A - Star Information:
Memory:

Close List Size:	1496
Open List Size:	558
Total Nodes Added:	2054
Total Nodes Generated:	11968
Total Nodes Generated But Not Added:	9914
Close List Sort:	0
Open List Sort:	1496

Quality:

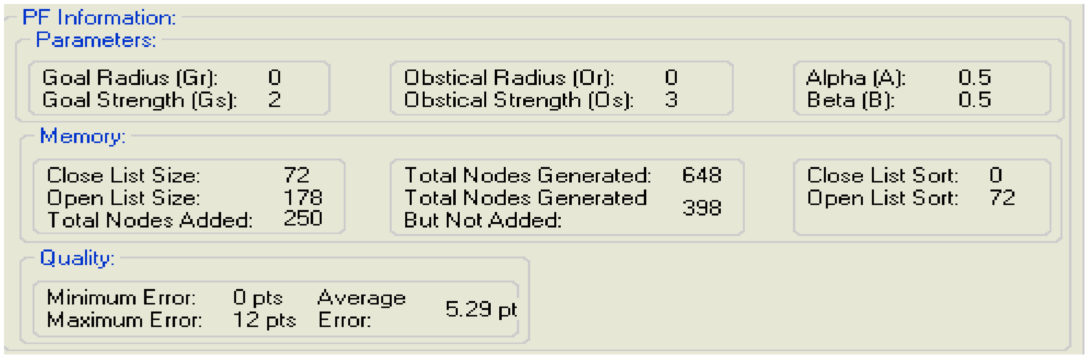
Minimum Error:	0 pts
Maximum Error:	20 pts
Average Error:	11.63 pt

RBFS Information:
Memory:

Close List Size:	263
Open List Size:	1249
Total Nodes Added:	1512
Total Nodes Generated:	2104
Total Nodes Generated But Not Added:	592
Close List Sort:	0
Open List Sort:	263

Quality:

Minimum Error:	0 pts
Maximum Error:	33 pts
Average Error:	16.64



PF Information:

Parameters:

Goal Radius (Gr):	0	Obstical Radius (Or):	0	Alpha (A):	0.5
Goal Strength (Gs):	2	Obstical Strength (Os):	3	Beta (B):	0.5

Memory:

Close List Size:	72	Total Nodes Generated:	648	Close List Sort:	0
Open List Size:	178	Total Nodes Generated But Not Added:	398	Open List Sort:	72
Total Nodes Added:	250				

Quality:

Minimum Error:	0 pts	Average Error:	5.29 pt
Maximum Error:	12 pts		

Figure 6.7: The information window showing the updated information about the AI algorithms

CHAPTER 7: EXPERIMENTS

In chapter 2 we tried to answer our first research question by identifying different AI problems and algorithms, and selecting some for our experiments. In this chapter we will conduct series of experiments using these selected problems and algorithms, and try to find answers of our remaining research questions. Firstly we will try to prove that making the AI algorithms anytime not only improve their performances but also provide a better solution for the AI problems especially in RTS gaming environments. Secondly we will compare the performances of these selected AI algorithms after making them anytime by using different performance measurements and environmental settings.

Thus, in this chapter we will conduct our experiments using our own built Path Finding Performance Comparison (PFPC) platform, followed by the results and the discussion for each of the experiment.

7.1 Experiment No. 01

In this experiment we will try to prove our claim that making AI algorithms anytime provides us with a better and more improved solution for the AI problems in RTS gaming environments.

7.1.1 Experimental Setup

The three AI algorithms that we will use for this experiment are A – Star (A*), Recursive Best First Search (RBFS) and Potential Fields (PF), as selected in chapter 2. And the three anytime algorithms that we will use for this experiment are anytime A* (AA*), anytime RBFS (ARBFS) and anytime PF (APF) as implemented in chapters 3, 4 and 5.

The AI problem for which we will compare the performances of these AI algorithms and their anytime counterparts is that of path finding, also selected in chapter 2.

The two performance measurements using those, we will compare the performances are that of Memory Usage (MU) and Quality of Results (QR). A brief description of what they are is provided below.

Memory Usage (MU) as the name suggests is the performance measurement using which we will determine the memory usage of any given AI algorithm. This will be one of our main performance measurements. As RTS gaming environments have some severe memory constraints, this performance measurement will help us better understand that how making AI algorithms anytime will improve and optimize their memory usage. The MU performance measurement is further sub divided into 8 fields, as listed below.

- CLS: This field holds the Close list’s size of any of the AI algorithm
- OLS: This field holds the Open list’s size of any of the AI algorithm
- TNA: This field holds the total nodes added in both the Close or Open lists of any of the AI algorithm
- TNG: This field holds the total nodes generated by any of the AI algorithm

- TNGnA: This field holds the total nodes generated but not added in either the Close or Open lists of any of the AI algorithm
- CLSo: This field holds the number of times Close list is been sorted by any of the AI algorithm
- OLSo: This field holds the number of times Open list is been sorted by any of the AI algorithm

Quality of Results (QR) is our second performance measurement that we will use to make performance comparison between our selected AI algorithms and their anytime counterparts. QR will show us the quality of the results that is been returned by any given AI algorithm during its search for the goal node. This performance measurement is important because it is crucial to know the quality of the results returned by any given AI algorithm in the RTS gaming environments.

The QR of any given AI algorithm is calculated by its deviation from the most optimal path possible. The most optimal path in any given environment is that of a straight line. That means an AI algorithm at best can do is to go straight from the root node to the goal node. But as we know, mostly this is not possible because there can possibly be many obstacles on the optimal path and the AI algorithm has to deviate from the optimal path to avoid them. What we have done here is to calculate this deviation of any given AI algorithm from the optimal path. The QR performance measurement is further sub divided into three fields as listed below.

- Min Err. This field holds the minimum error (deviation) of any of the AI algorithm
- Max Err. This field holds the maximum error of any of the AI algorithm
- Avg Err. This field holds the average error of any of the AI algorithm

Now towards the end we will discuss the environmental settings that we will use for this experiment. A discussed earlier in chapter 6, a PFPC user can load a map into the environment using many different settings and options. We will use four among those settings, as listed below.

• Area Covered 30%	-- Fixed Sized (8 * 8)	-- Separated
• Area Covered 20%	-- Fixed Sized (8 * 8)	-- Separated
• Area Covered 30%	-- Variable Sized (4 * 4) – (8 * 8)	-- Separated
• Area Covered 30%	-- Fixed Sized (8 * 8)	-- Random/Mixed

A total of 200 maps will be used, 50 for each environmental setting. Each result for each AI algorithm is the average of the results from all the 200 maps used. Towards the end it is important to mention that these performance measurements or environmental settings if used in the up coming experiments will not be discussed again.

7.1.2 Experimental Results

Listed below are the results of our selected AI algorithms and their anytime counterparts using the MU as performance measurement.

	CLS	***OLS***	***TNA***	***TNG***	***TNGnA***	***CLSo***	***OLSo***
A*	733.10	488.52	1221.62	5864.80	4643.17	0	733.10
RBFS	151.78	860.93	1012.71	1215.40	202.68	0	151.92
PF	187.53	245.49	433.03	1687.81	1254.57	0	187.53

Table 7.1: Results of our AI algorithms using MU as parameter measurement

	CLS	***OLS***	***TNA***	***TNG***	***TNGnA***	***CLSo***	***OLSo***
AA*	190.64	419.81	610.45	1533.80	922.26	0	191.72
ARBFS	151.33	859.59	1010.92	1211.84	200.91	0	151.48
APF	206.00	243.83	449.83	1691.05	1259.12	0	187.89

Table 7.2: Results of our anytime AI algorithms using MU as parameter measurement

Listed below are the results of our selected AI algorithms and their anytime counterparts using the QR as the performance measurement.

	Min Err.	***Max Err.***	***Avg Err.***		***Min Err.***	***Max Err.***	***Avg Err.***
A*	0	22.615	11.432	**AA***	0	24.210	12.016
RBFS	0	25.365	12.504	**ARBFS**	0	25.390	12.513
PF	0	22.395	9.062	**APF**	0	22.255	9.073

Table 7.3: Results of our AI algorithms and their anytime counterparts using QR as parameter measurement

7.1.3 Experiment Discussion

For the first part of our discussion we will compare the performances of our selected AI algorithms and their anytime counterparts using the MU performance measurement. Shown below are the above listed results of our AI algorithms using MU performance measurement in the form of a bar chart.

- Consider the first three fields in the bar chart. What we can see is that the number of nodes in the Open list (OLS) for the RBFS algorithm is greater than A* or PF algorithms. This is because the RBFS algorithm didn’t have any checks on the redundant nodes, except those on the current path. Due to this the RBFS algorithm adds many nodes several times, which increases its Open list size.

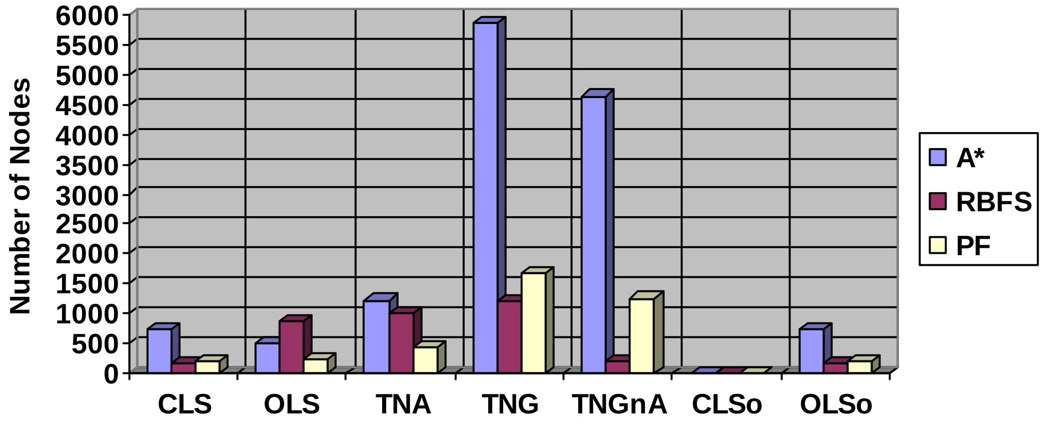


Figure 7.1: Bar chart showing results of our AI algorithms using MU as performance measurement

- However if we see the number of total nodes added (TNA) in both the Close and the Open lists for the RBFS algorithm, it is less than that of the A* algorithm. This is because although A* algorithm has lesser number of nodes in the Open list but it has much larger number of nodes in the Close list as compare to the RBFS algorithm. One possible explanation for this is how the A* algorithm handles the local minima. As shown in chapter 3, whenever A* algorithm hits the local minima it adds not only those nodes in the local minima but also those in between the local minima and the root node into the Close list. The RBFS algorithm on the other hand deals much better with the local minima by adding nodes, only those are in the local minima. That is why the number of nodes in the Close list for the RBFS algorithm is much less than that of the A* algorithms.
- PF algorithm on the other hand has much lesser number of total nodes added in the Close and the Open lists as compare to both the A* and the RBFS algorithms. This is because the A* and the RBFS algorithms use these lists for the systematic search to find the goal node. PF algorithm on the other hand uses these lists only to deal with the local minima.
- Now consider the total nodes generated (TNG) and total nodes generated but not added (TNGnA) fields of all the three AI algorithms. The A* algorithm generates lots of nodes as compare to the RBFS and the PF algorithms. As shown in chapter 3, this is because how it deals with the local minima situation by generating nodes all the way back to the root node which increases its node generation exponentially.
- Although the A* and the PF algorithm have higher nodes generation as compare to the RBFS algorithm but in TNGnA field what we can see is that the most of the nodes generated by the A* and the PF are never added, resulted in a much less Open list size. But as the RBFS algorithm didn’t have any redundancy check most of its generated nodes are added into the Open list. This can be seen by its large Open list size.
- The number of times the Open list sorts (OLSo) for any of the AI algorithm is closely matched with the size of its Close list (CLS). This is because every time an AI algorithm adds a node in the Close list, it adds its successor nodes in the Open list and thus Open list needs to be sorted accordingly.

Shown below are the above listed results of our anytime AI algorithms using MU performance measurement in the form of a bar chart. It shows how the MU performance of our AI algorithms changes when converted into anytime. It is important to mention here that the scale used in both the bar charts is the same.

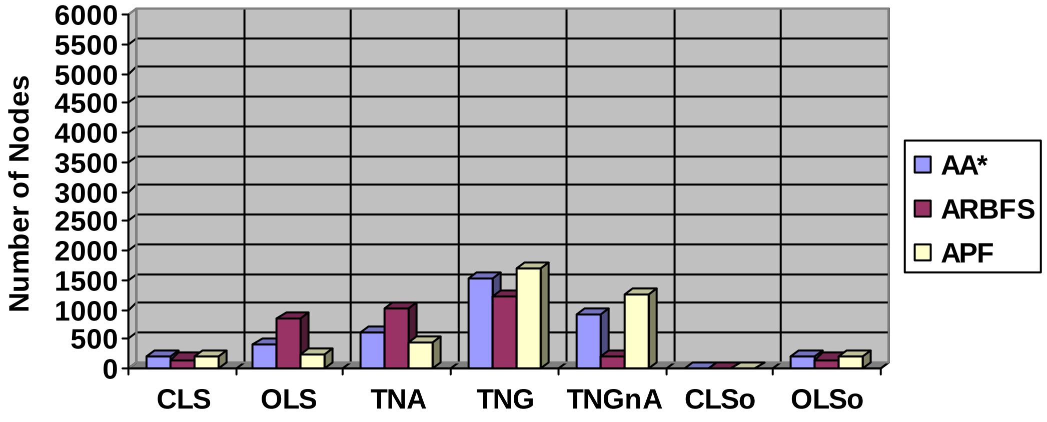


Figure 7.2: Bar chart showing results of our anytime AI algorithms using MU as performance measurement

- The most obvious thing that is noticeable when we compare both the bar charts, is this huge reduction in the total nodes generated (TNG) by our AA* algorithm. The total nodes generated by the AA* algorithm is much lesser than the A* algorithm. This is because of how our implemented AA* algorithm deals with the local minima. This can more suitably be explained with an example below.

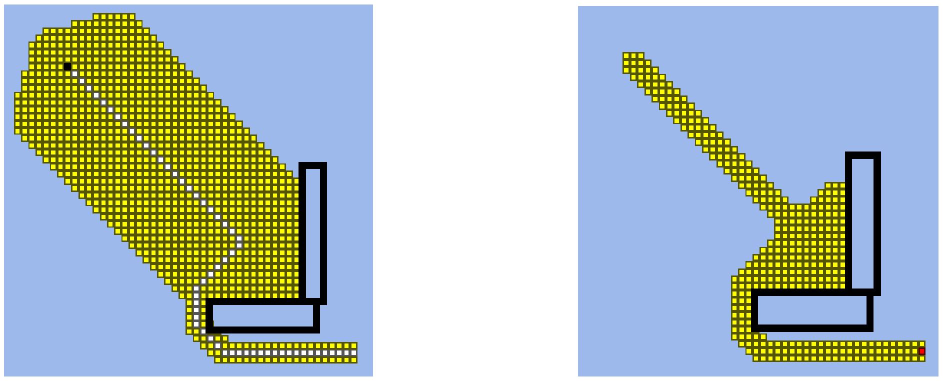


Figure 7.3: How our anytime A algorithm deals with local minima as compare to A* algorithm*

The way our AA* algorithm handles the local minima is the main reason for the reduction in the total nodes generation. This reduction of the total nodes generated also has its effect on the Close (CLS)and the Open (OLS) lists as well as on the total nodes added (TNA) by the AA* algorithm. By close observation we can see that the size of the Close list reduces much more than that of the Open list. This also proves our point from the discussion of the previous bar chart, relating total nodes generated by the A* algorithm and its direct effect on its Close list size.

- The MU performance of the remaining two algorithms, ARBFS and APF, remains very much the same when converted into anytime algorithms. The reason for that is how ARBFS and APF handle the local minima remains pretty much the same as that of the RBFS and the PF algorithms. However by close observation we can see that the memory usage of the PF algorithm increases slightly when converted into anytime algorithms. This is due to the overhead that anytime algorithms have due to the anytime property of interrupt-ability. When we stop and then restart an anytime algorithm some nodes are regenerated because when stopped previously the anytime algorithm stores only the best node and removes all the remaining nodes from the Open and the Close lists.
- A slight increase in the memory usage of the APF algorithms doesn't mean that converting these algorithms into anytime is not a good idea; because we are missing one important point that anytime algorithms never store all the nodes in the memory. When an anytime algorithm stops, it clears all its memory and stores only the best node. AI algorithms on the other hand have to store all the nodes in the memory all the time until they find the goal node. In the memory constrained environments of RTS gaming anytime algorithms can easily out perform the AI algorithms.

Now we will consider Quality of Results (QR) performance measurement to compare the performances of the AI algorithms and their anytime counterparts.

- If we compare the results of AI algorithms and their anytime counterparts using the QR performance measurement, it is noticeable that after converting these algorithms into anytime, their quality of results decreases. This decrease in their quality of results is biggest in the case of AA* algorithm. The reason for this decrease in quality is because how anytime algorithms store their information (nodes) about their environments. Anytime algorithms when stopped clear their Open and Close lists, thus taking risk of losing some important environmental information. When restarted anytime algorithms have no information about their previous runs except that of the best node. AI algorithms on the other hands store all the information (nodes) in the memory all the time therefore they have all the information they required in order to select the optimal path to the goal node.

In the remaining series of experiments we will try to compare the performances of our three anytime AI algorithms with respect to each other under various environmental settings.

7.2 Experiment No. 02

In this first experiment of a series of experiments we will try to analyze the effect of total area covered by the obstacles in an environment, on the performances of our three anytime AI algorithms.

7.2.1 Experimental Setup

The three anytime algorithms that we will use for this experiment are AA*, ARBFS and APF and the AI problem for which we will compare their performances, is that of path finding.

The two performance measurements using those we will compare the performances of these anytime algorithms are Memory Usage (MU) and Quality of Results (QR). These both are discussed well in detail in the previous experiment.

The environmental settings that we will use for this experiment are listed below.

- Area Covered 30% -- Fixed Sized (8 * 8) -- Separated
- Area Covered 20% -- Fixed Sized (8 * 8) -- Separated

We have increased the total area covered by the obstacles in an environment by 10% and now we will see what effect it has on the performances of our three anytime AI algorithms. A total of 100 maps will be used in this experiment, 50 for each environmental setting. Each result for each algorithm is the average of the results from all the 100 maps used.

7.2.2 Experimental Results

Listed below are the results of our three anytime AI algorithms for both the 30% and the 20% of the total area covered by the obstacles using the MU as performance measurement.

30%	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	217.10	425.96	643.06	1746.24	1102.00	0	218.28
ARBFS	157.68	884.56	1042.24	1262.56	220.32	0	157.82
APF	176.96	223.22	400.18	1592.64	1192.28	0	176.96

Table 7.4: Results of our anytime AI algorithms when 30% of the area covered by the obstacles using MU as performance measurement

20%	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	166.60	409.44	576.04	1340.80	763.76	0	167.60
ARBFS	139.12	808.00	947.12	1113.12	166.00	0	139.14
APF	122.92	265.78	388.70	1106.28	717.30	0	122.92

Table 7.5: Results of our anytime AI algorithms when 20% of the area covered by the obstacles using MU as performance measurement

Listed below are the results of our three anytime AI algorithms for both the 30% and the 20% of the total area covered by the obstacles, now using the QR as performance measurement.

30%	*Min Err.*	*Max Err.*	*Avg Err.*	*20%*	*Min Err.*	*Max Err.*	*Avg Err.*
AA*	0	24.36	12.0764	**AA***	0	24.02	11.9122
ARBFS	0	25.18	12.1924	**ARBFS**	0	24.98	12.1494
APF	0	22.5	9.3642	**APF**	0	15.66	5.9566

Table 7.6: Results of our anytime AI algorithms for both 30% & 20% of the area covered by the obstacles using QR as performance measurement

7.2.3 Experiment Discussion

Shown below are the two bar charts of the results of our three anytime AI algorithms for both the 30% and the 20% of the total area covered by the obstacles using the MU as performance measurement.

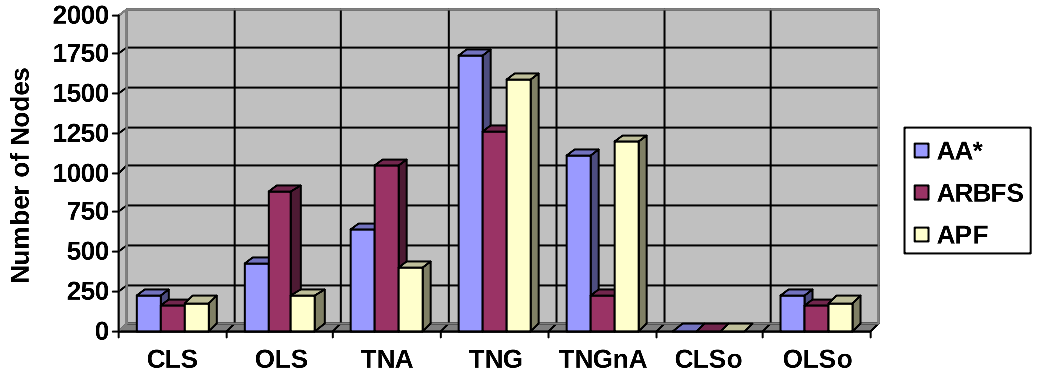


Figure 7.4: Bar chart showing the results of our anytime AI algorithms when 30% of the area covered by the obstacles using MU as performance measurement

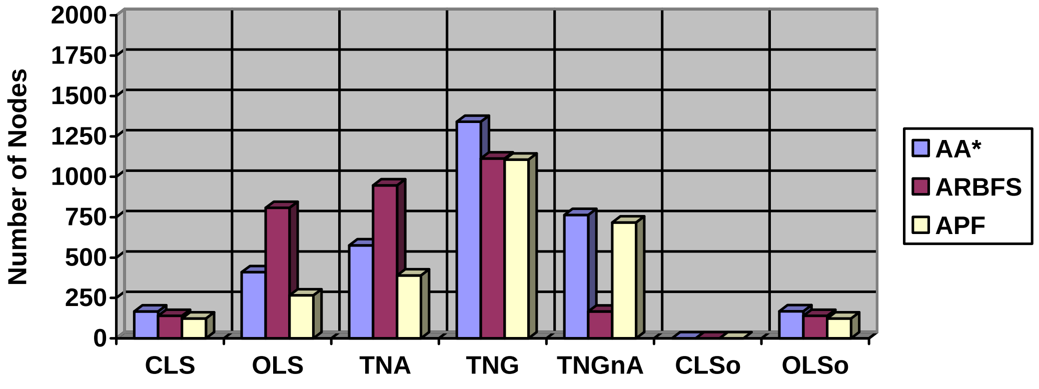


Figure 7.5: Bar chart showing the results of our anytime AI algorithms when 20% of the area covered by the obstacles using MU as performance measurement

- If we compare the results of our three anytime AI algorithms, it is noticeable that by reducing the total area covered by the obstacles the overall memory usages of all of the three anytime AI algorithms have reduced. The reduction in the total area covered by the obstacles in an environment means lesser number of obstacles in the environment; thus anytime AI algorithms find it easy to find their way towards the goal node with lesser node generation.
- By close observation we can see that unlike AA* and ARBFS algorithms the size of the Open list (OLS) for the APF algorithm has increased. This increase is due to the fact that how APF uses its Open list. APF normally generates new nodes simply by calculating the combined potential fields of all the objects for every point in the environment. These new nodes are then added into the Close list and all their successor nodes into the Open list. APF algorithm only uses these successor nodes in case of the local minima. The reduction in the total area covered by the obstacles means that APF will encounter lesser number of local minima and thus lesser need of the Open list. Due to this lesser usage of the Open list results in increase of its size.
- The total node generated (TNG) and the total node generated but not added (TNGnA) fields of all the three anytime AI algorithms, show some serious reduction in their sizes simply because of lesser number of obstacles and easy to find goal node.

Now we will consider Quality of Results (QR) performance measurement to compare the performances of the three anytime AI algorithms.

- If we compare the results of our three anytime AI algorithms, it is noticeable that by reducing the total area covered by the obstacles the overall quality of the results for all of the three anytime AI algorithms have improved.
- Unlike AA* and ARBFS algorithms, the improvement in quality is biggest for the APF algorithm. This is because of the two main reasons; Firstly in case of APF algorithm each obstacles is surrounded with a repelling potential which can deviate APF algorithm much further away from the optimal path as compared to AA* and ARBFS algorithms. But now with lesser number of obstacles APF algorithm can find goal node with minimum deviation from optimal path. Secondly, APF algorithm has an advantage over the heuristic based algorithms. Due to their implemented heuristic, AA* and ARBFS algorithms cannot always take straight line path towards the goal as compare to APF algorithm which always goes directly towards the goal guided by the potential fields.

7.3 Experiment No. 03

In this second experiment of a series of experiments we will try to analyze the effect of the sizes of the obstacles in an environment, on the performances of our three anytime AI algorithms.

7.3.1 Experimental Setup

The three anytime algorithms that we will use for this experiment are AA*, ARBFS and APF and the AI problem for which we will compare their performances, is that of path finding.

The two performance measurements using those we will compare the performances of these anytime algorithms are Memory Usage (MU) and Quality of Results (QR).

The environmental settings that we used for this experiment are listed below.

- Area Covered 30% -- Fixed Sized (8 * 8) -- Separated
- Area Covered 30% -- Variable Sized (4 * 4) – (8 * 8) -- Separated

We have changed the fixed sizes of the obstacles in an environment into the variable sizes and now we will see what effect it has on the performances of our three anytime AI algorithms. A total of 100 maps will be used in this experiment, 50 for each environmental setting.

7.3.2 Experimental Results

Listed below are the results of our three anytime AI algorithms for both the fixed and the variable sizes of the obstacles using the MU as performance measurement.

Fixed	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	217.10	425.96	643.06	1746.24	1102.00	0	218.28
ARBFS	157.68	884.56	1042.24	1262.56	220.32	0	157.82
APF	176.96	223.22	400.18	1592.64	1192.28	0	176.96

Table 7.7: Results of our anytime AI algorithms when obstacles are of fixed sized using MU as performance measurement

Variable	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	183.48	421	604.48	1476.32	870.78	0	184.54
ARBFS	148.96	853.18	1002.14	1193.28	191.14	0	149.16
APF	301.5	224.62	526.12	2391.84	1901.28	0	265.76

Table 7.8: Results of our anytime AI algorithms when obstacles are of variable sized using MU as performance measurement

Listed below are the results of our three anytime AI algorithms for both the fixed and the variable sizes of the obstacles using the QR as performance measurement.

Fixed	*Min Err.*	*Max Err.*	*Avg Err.*	*Variable*	*Min Err.*	*Max Err.*	*Avg Err.*
AA*	0	24.36	12.0764	**AA***	0	22.9	11.2294
ARBFS	0	25.18	12.1924	**ARBFS**	0	24.68	12.2692
APF	0	22.5	9.3642	**APF**	0	26.36	10.7986

Table 7.9: Results of our anytime AI algorithms when obstacles are of both fixed and variable sized using QR as performance measurement

7.3.3 Experiment Discussion

Shown below are the two bar charts of the results of our three anytime AI algorithms for both the fixed and variable sized obstacles, using the MU as performance measurement.

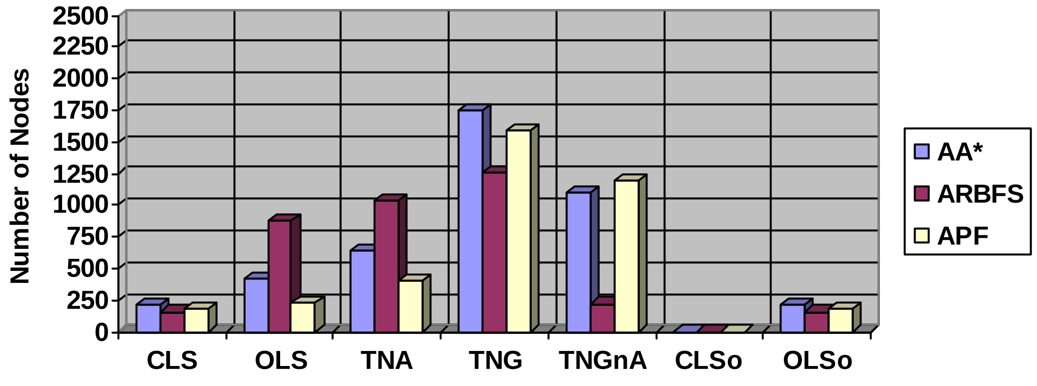


Figure 7.6: Bar chart showing the results of our anytime AI algorithms when obstacles are of fixed sized using MU as performance measurement

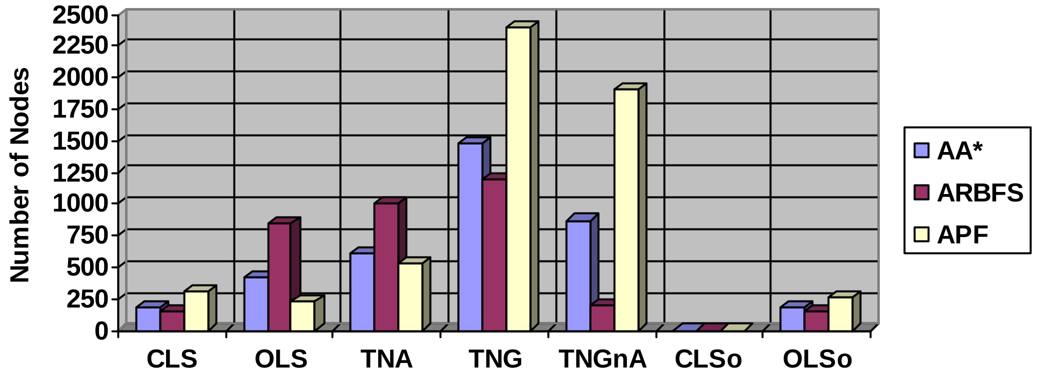


Figure 7.7: Bar chart showing the results of our anytime AI algorithms when obstacles are of variable sized using MU as performance measurement

- If we compare the results of our three anytime AI algorithms, it is noticeable that by converting to variable sized obstacles the overall memory usages for AA* and ARBFS algorithms have reduced. When we converted the fixed size obstacles into variable sized obstacles and try to keep the total area covered by these obstacles to 30%, the number of obstacles in the environment has increased. The above results have shown us that the AA* and the ARBFS algorithms are more memory efficient even with the larger number of variable sized obstacles as compare to the smaller number of fixed sized obstacles.

- However in case of APF algorithm, the memory usage has increased a lot when converted to variable sized obstacles. With the increase in the number of obstacles APF algorithm has to generate lots of nodes to get around them and find the goal node. Even with this increase in memory usage, APF algorithm still has lesser number of nodes added in both the Open and the Close lists as compare to AA* and ARBFS algorithm.

Now we will consider Quality of Results (QR) performance measurement to compare the performances of the three anytime AI algorithms.

- If we compare the results of our three anytime AI algorithms, it is noticeable that by converting to variable sized obstacles the overall quality of the results for ARBFS and APF algorithms have reduced. As also explained in the previous experiment that due to the larger number of obstacles in the environment, these algorithms have to deviate away from the optimal path to reach the goal node. However in case of AA* algorithm the quality of results has improved unexpectedly.
- Also from the previous experiment in which the quality of results for APF algorithm improves significantly with decrease in the number of obstacles. What we are expecting here is that with the increase in the number of obstacles in this experiment the quality of results for APF algorithm will reduce with the same rate. But what we have seen here is that its quality decreases slightly.

7.4 Experiment No. 04

In this third and the last experiment of the series of experiments we will try to analyze the effect of the layouts of the obstacles in an environment, on the performances of our three anytime AI algorithms.

7.4.1 Experimental Setup

The three anytime algorithms that we will use for this experiment are AA*, ARBFS and APF and the AI problem for which we will compare their performances, is that of path finding.

The two performance measurements using those we will compare the performances of these anytime algorithms are Memory Usage (MU) and Quality of Results (QR).

The environmental settings that we used for this experiment are listed below.

• Area Covered 30%	-- Fixed Sized (8 * 8)	-- Separated
• Area Covered 30%	-- Fixed Sized (8 * 8)	-- Random/Mixed

We have changed the separated layout of the obstacles in an environment into the random/mixed layout and now we will see what effect it has on the performances of our three anytime AI algorithms. A total of 100 maps will be used in this experiment, 50 for each environmental setting.

7.4.2 Experimental Results

Listed below are the results of our three anytime AI algorithms for both the separated and the random/mixed layouts of the obstacles using the MU as performance measurement.

Separate	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	217.10	425.96	643.06	1746.24	1102.00	0	218.28
ARBFS	157.68	884.56	1042.24	1262.56	220.32	0	157.82
APF	176.96	223.22	400.18	1592.64	1192.28	0	176.96

Table 7.10: Results of our anytime AI algorithms with separated layout of the obstacles using MU as performance comparison

Mixed	*CLS*	*OLS*	*TNA*	*TNG*	*TNGnA*	*CLSo*	*OLSo*
AA*	195.38	422.86	618.24	1571.84	952.5	0	196.48
ARBFS	159.58	892.62	1052.2	1278.4	226.2	0	159.8
APF	222.62	261.72	484.34	1673.46	1225.62	0	185.94

Table 7.11: Results of our anytime AI algorithms with mixed/random layout of the obstacles using MU as performance comparison

Listed below are the results of our three anytime AI algorithms for both the separated and the random/mixed layouts of the obstacles using the QR as performance measurement.

Fixed	*Min Err.*	*Max Err.*	*Avg Err.*	*Variable*	*Min Err.*	*Max Err.*	*Avg Err.*
AA*	0	24.36	12.0764	**AA***	0	25.56	12.8474
ARBFS	0	25.18	12.1924	**ARBFS**	0	26.72	13.4424
APF	0	22.5	9.3642	**APF**	0	24.5	10.1726

Table 7.12: Results of our anytime AI algorithms with both separated and mixed/random layouts of the obstacles using QR as performance comparison

7.4.3 Experiment Discussion

Shown below are the two bar charts of the results of our three anytime AI algorithms for both the separated and the random/mixed layout of the obstacles, using the MU as performance measurement.

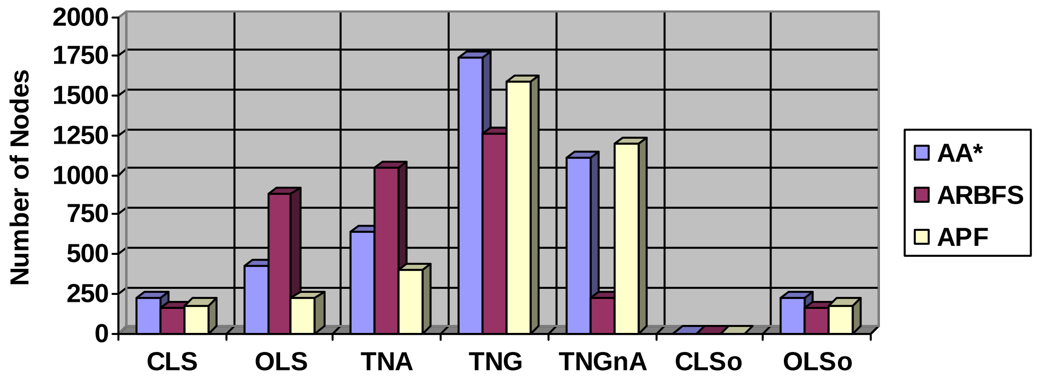


Figure 7.8: Bar chart showing the results of our anytime AI algorithms with separated layout of the obstacles using MU as performance comparison

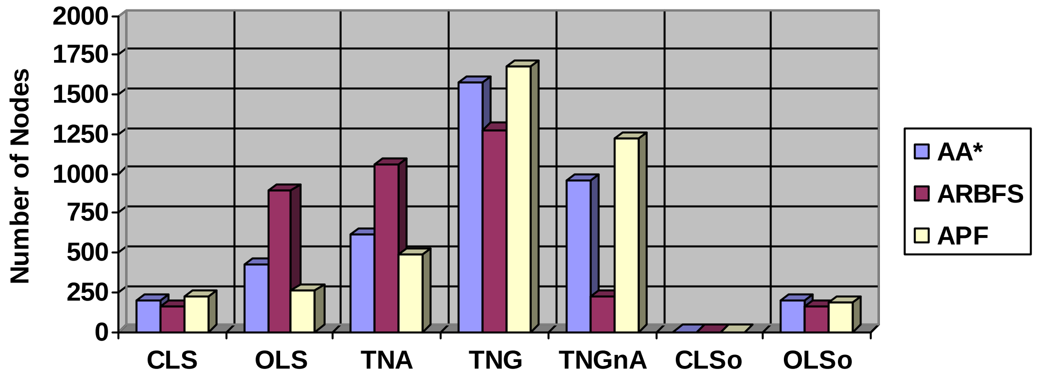


Figure 7.9: Bar chart showing the results of our anytime AI algorithms with mixed/random layout of the obstacles using MU as performance comparison

- If we compare the results of our three anytime AI algorithms, it is noticeable that by converting to random/mixed layout the overall memory usages for AA* algorithm has reduced. For APF algorithm the memory usage has increased and for ARBFS algorithm the memory usage remains pretty much the same.

Now we will take our second performance measurement of quality of results and show how it is affected when converted from separated to mixed/random layout.

- If we compare the results of our three anytime AI algorithms, it is noticeable that by converting to random/mixed layout the overall quality of results for all the three anytime algorithms have reduced.
- When converted from the separated to the random/mixed layout of the obstacles, it is noticeable that obstacles are combined together to form complex large size obstacles. These obstacles then force our three anytime AI algorithms to deviate away from the optimal path.

CHAPTER 8: DISCUSSION

In previous chapter we have carried out series of experiments to answer our two main research questions. Firstly we tried to prove that by making AI algorithms anytime provides us with a better and more improved solution for our AI problems in RTS gaming environments. And secondly we compared the performances of our three anytime algorithms (AA*, ARBFS and APF) in search of a possible solution for our AI problem (Path finding).

In this chapter we will discuss the results from our experiments; although each experiment has already been discussed well in detail along side its results in the previous chapter but in this chapter we will try to put all these discussions into perspective and try to figure out that by conducting these experiments did we achieve our required goals or not?

8.1 Discussion about Results:

The first of our experiment was regarding the performance comparison between three AI algorithms of A*, RBFS and PF and their anytime counterparts of AA*, ARBFS and APF algorithms. What we have tried to prove by this experiment is that by converting these AI algorithms into anytime algorithms have improved their performances. The performance measurements that we have used are that of Memory Usage (MU) and Quality of Results (QR). The results from this experiment are discussed below

- When compared using the MU performance measurement the AA* algorithm has shown the huge memory reduction as compared to the A* algorithm. This is simply because how our AA* algorithm deals with the local minima. But in the case of APF, its total memory usage has increased when compared with the PF algorithm. This is because of the overhead that anytime algorithms have due to their anytime property of interrupt-ability.

- On the other hand when compared using the QR as performance measurement, the quality of the results for all the three anytime algorithms have decreased as compared to the AI algorithms. This is because whenever our anytime algorithms stop, they clear their Open and the Close lists, result in the loss of some important environmental information (nodes). The AI algorithms on the other hand store all the environmental information (nodes) in the memory all the time and thus able to select the optimal path towards the goal node.

- However this doesn’t mean that converting AI algorithms into anytime algorithms is a bad idea because we are missing one important point here is that the anytime algorithms never always store all the nodes in the memory. When stop and then restart, anytime algorithms clear all the nodes from the Open and the Close lists. Therefore although their total memory usages are greater than that of the AI algorithms but anytime algorithms at one time have much lesser number of nodes in the memory. The AI algorithms on the other hand have to store all the nodes in the memory all the time. When consider this point, anytime algorithms are much more memory efficient as compared to the AI algorithms.

- But here we hit a dilemma situation. Anytime algorithms are much more memory efficient simply because they clear their Open and Close lists. when stop, and restarts off with the fresh ones, but due to this clearing of the Open and the Close lists they tend to lost some important information about their environment thus result in a poor quality of results as compare to the AI algorithms.

This experiment however is an overall success because by conducting this experiment we better understand the disadvantages and advantages of converting AI algorithms into anytime algorithms.

Some of the disadvantages are listed below

- The quality of results of all the AI algorithms have reduced when converted to anytime algorithms. However if consider it statistically (*AA* 0.58%, ARBFS 0.01%, APF 0.01%*) this reduction is very negligible.
- Conversion of AI algorithms into anytime algorithms has some overhead associated with it that can result in increase of the total memory usage of some of the anytime algorithms (APF).

Some of the advantages are listed below

- The three anytime algorithms are much more memory efficient as compare to the AI algorithms. This can boost their performance considerably as compare to the AI algorithms in RTS gaming environments.
- The three anytime algorithms have also helped us to overcome the time constraints of RTS gaming environments. As we know that AI algorithms have to run till the end to output any results that make them unsuitable for RTS gaming environments. Our three anytime algorithms can be stop and then restart at anytime.

The remaining three of our experiments were regarding the performance comparison between our three anytime algorithms of AA*, ARBFS and APF. What we have tried to analyze by these experiments is that which anytime algorithm has performed better with respect to the other two anytime algorithms under which environmental settings. The performance measurements that we have used are that of Memory Usage (MU) and Quality of Results (QR). The results from these experiments are discussed below

- In the first among these experiment we have tried to see the effect of total area covered by the obstacles in an environment on the performances of our three anytime algorithms. The results from this experiment have shown that all the three anytime algorithms have reduced their memory usages and their quality of results have improved when total area covered by the obstacles has reduced.

- In the second among these experiment we have tried to see the effect of different sized obstacles on the performances of our three anytime algorithms. The results from this experiment have shown that the memory usages of the

AA* and the ARBFS algorithms have reduced when converted from the fixed sized obstacles to the variable sized ones. This shows that the AA* and ARBFS algorithms can still perform better with relatively greater number of small sized obstacles as compare to the lesser number of big sized obstacles. However in case of the APF algorithm, its memory usage has increased significantly. This shows that the increase in the number of obstacles though small in number in an environment can effect APF more than other two anytime algorithms. The quality of results of AA* algorithm has improved and that of ARBFS and APF have reduced slightly when converted to variable sized obstacles.

- In the third and the last among these experiments we have tried to see the effect of different layouts of the obstacles on the performances of our three anytime algorithms. The results from this experiment have shown that the memory usages of AA* and ARBFS algorithms have reduced or remained approximately the same when converted from separated layout to the random/mixed layout. However for the APF algorithm its memory usage has increased. The quality of results of all the three anytime algorithms have reduced when converted to random/mixed layout.

These three experiments are also a success as they helped us better understand and analyze the performances of our three anytime AI algorithms under various environmental settings using different performance measurements. However even after detailed analysis it is hard for us to reach on a decision to declare any one among these three anytime algorithms as the best algorithm. What we can do here however is to declare any one anytime algorithm as the best performed algorithm under a specified environmental settings.

Listed below are some points of our conclusion that we have reached after our analysis of the results of these three experiments.

- Firstly if we consider the QR performance measurement, it has been noticed that the quality of results of the APF algorithm remains the best as compare to the other two anytime algorithms of AA* and ARBFS, no matter what the environmental settings are for the AI problem of path finding. So we can safely conclude that in all those RTS environments in which quality of results is of highest priority, APF algorithm is the best choice among the three anytime algorithms we have discussed here.
- However the situation is very vague as far as MU performance measurement is concerned, because not one anytime algorithm can be declared as the best perform algorithm under all environmental settings.
- The hardest part is to correctly judge the memory usage of the APF algorithm as it can change significantly with the slight change in the environmental settings. However memory usage of APF is much less than AA* and ARBFS algorithms when there are lesser number of obstacles in the environment (*lesser total area covered by the obstacles*). If the number of obstacles in the environment increases the memory usage of APF also increases significantly as compare to the AA* and ARBFS algorithms. So what we have concluded here is that the APF algorithm is only suitable in those environments with lesser number of obstacles.

- However as far as AA* and ARBS algorithms are concerned, unlike APF their memory usages remain reasonable and there is no significant rise or fall in these. That makes these two anytime algorithms suitable for most environmental settings. If we compare the memory usages of the AA* and ARBFS algorithms with respect to each other then in ARBFS algorithm the total node generation remains always less as compare to the AA*. So ARBFS as the memory efficient algorithm of AA* is the preferable selection if we have to select any one among two of these.

CONCLUSION

The purpose of this study is to identify different AI algorithms and problems in RTS gaming, and to analyze whether making AI algorithms anytime is a possible solution for these AI problems or not? And then finally we need to compare the performances of different anytime algorithms using our own platform for an AI problem.

- During our study we have achieved all our objectives. In chapter 2 we have not only identify different AI algorithms and problems but also discuss them in brief and then select some for our experimentation later on.

- In chapter 7 with the help of the results from our first experiment we have seen that making AI algorithms anytime, helps these algorithms to better consume their memory and time resources in the RTS environments. Although making AI algorithms anytime also have some disadvantages but they are negligible as compare to the advantages it have.

- Then also in chapter 7 with the help of the results from our remaining experiments we have compared the performances of our anytime algorithms under various performance measurements and environmental settings. As far as QR performance measurement is concerned APF has performed better than the AA* and ARBFS algorithms under all environmental settings.

- However as far a MU performance measurement is concerned, no direct selection is possible as no anytime algorithm has performed best under all the environmental settings. APF has performed best only with lesser number of obstacles in the environment but as the number of obstacles increased the memory usage of APF also increased significantly. However the memory usage of AA* and ARBFS algorithms remained pretty much the same under various environmental settings. ARBFS algorithm being the memory efficient algorithm than AA* is preferable in that case.

FUTURE WORK

With this study we have prove that making AI algorithms anytime can provide us with better and more improved solution for AI problems in the RTS gaming environments. Also we analyze and compare the performances of different anytime algorithms under various environmental settings and decide which anytime algorithm has performed better than the others under which environmental settings.

However there are certain things that we wish to perform in the future to further refine our current study.

- APF algorithm has a certain set of parameters.
  Parameters like
  - Goal Radius:
    This parameter holds the radius of the goal node.
  - Goal Strength:
    This parameter holds the strength of the potential field with which the goal is attracting.
  - Obstacle Radius:
    This parameter holds the radius of an obstacle.
  - Obstacle Strength:
    This parameter holds the strength of the repelling force with which the obstacle is repelling.

  What we have noticed that memory usage of APF algorithm can vary widely with slight variation in its parameters. What we would like to do in the future is to test our APF algorithm with different values of its parameters and check how it effects its memory usage as compare to the other two anytime algorithms of AA* and ARBFS.

- In the study so far we have compared the performances of our anytime algorithms using only the MU and QR as the performance measurements. But what we would like to do in the future is to use Computational Time (CT) as our third performance measurement as it would be important to know the performances of our anytime algorithms using this performance measurement.

- Also we would like to add new algorithms into our PFPC platform, make them anytime and then compare their performances using different performance measurements and environmental settings.

REFERENCES

[1] M. Buro and T. Furtak
"RTS Games and Real-Time AI Research"
Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS), Arlington VA 2004

[2] V. Honavar
"Artificial Intelligence: An Overview"
AI Research Laboratory- Department of Computer Science- Iowa State University- Ames 2006

[3] J. David
"Artificial Intelligence for Computer Games: An Introduction"
Publisher: AK Peters, Ltd. 2004, Pages: 146, ISBN: 15-6881-208-6

[4] M. Mateas
"Expressive AI: Games and Artificial Intelligence"
The Georgia Institute of Technology- Atlanta 2003

[5] M. Buro and T. Furtak
"On the Development of a Free RTS Game Engine"
Department of Computer Science- University of Alberta 2005

[6] S. Russell and P. Norvig
"Artificial Intelligence: A Modern Approach- Second Edition"
Publisher: Pearson Education, Inc. 2003, Pages: 1050, ISBN: 81-7758-367-0

[7] K. Fujisawa, S. Hayakawa, T. Aoki, T. Suzuki and S. Okuma
"Real Time Motion Planning for Autonomous Mobile Robot using Framework of Anytime Algorithm"
IEEE- International Conference on Robotics & Automation Detroit, Michigan, 1999

[8] W. Kywe, D. Fujiwara and K. Murakami
"Scheduling of Image Processing Using Anytime Algorithm for Real-time System"
IEEE- International Conference on Pattern Recognition, 2006

[9] A. Hansen, R. Zhou
"Anytime Heuristic Search"
Journal of Artificial Intelligence Research, 2006

[10] T. Nurkkala
"Informed Search – Lecture 4"
Department of Computer Science, University of Minnesota,
http://www-users.itlabs.umn.edu/classes/Spring-2007/csci5511/ pdf/04-slides.pdf, 2007

[11] J. Laird, M. Lent
"Human- level AI's Killer Application: Interactive Computer Games"
University of Michigan, 2001

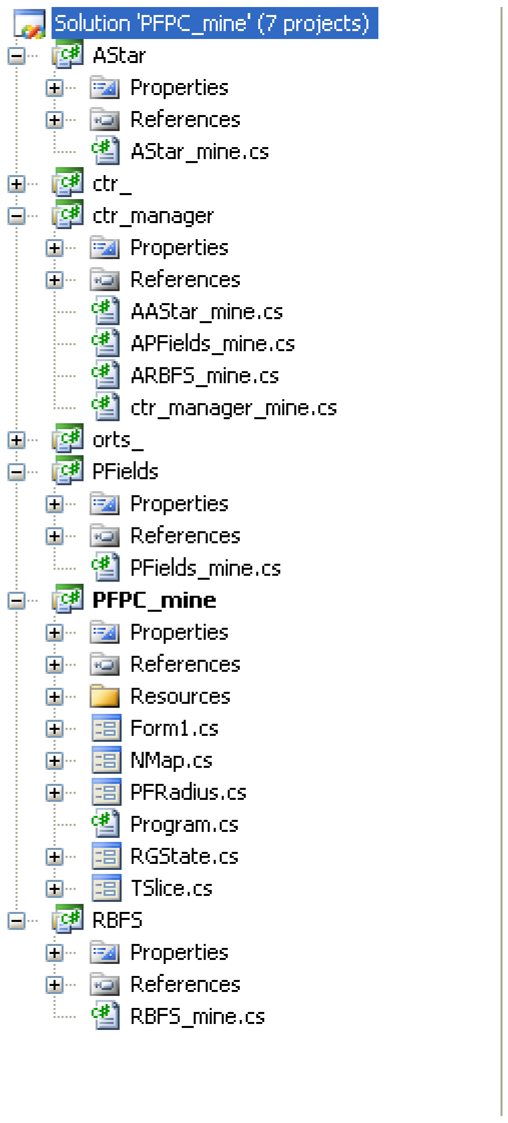
[12] J. Orkin
"Agent Architecture Considerations for Real-Time Planning in Games"
AIDE Proceedings, 2005

[13] M. Goodrich
"*Potential Fields Tutorial"*
http://students.cs.byu.edu/~cs470ta/readings/Pfields.pdf, 2002

[14] H. Cazorla
"*Multiple Potential Fields in Quake 2, Multiplayer"*
Blekinge Institute of Technology, Aug 2006.

[15] S. Zilberstein
"*Using Anytime Algorithms in Intelligent Systems"*
1996

APPENDIX A: PFPC STRUCTURE



```
Solution 'PFPC_mine' (7 projects)
AStar
    Properties
    References
    AStar_mine.cs
ctr_
ctr_manager
    Properties
    References
    AAStar_mine.cs
    APFields_mine.cs
    ARBFS_mine.cs
    ctr_manager_mine.cs
orts_
PFields
    Properties
    References
    PFields_mine.cs
PFPC_mine
    Properties
    References
    Resources
    Form1.cs
    NMap.cs
    PFRadius.cs
    Program.cs
    RGState.cs
    TSlice.cs
RBFS
    Properties
    References
    RBFS_mine.cs
```