MULTI-OBJECTIVE PATH PLANNING FOR VIRTUAL ENVIRONMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

TUĞCEM ORAL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2012

Approval of the thesis:

MULTI-OBJECTIVE PATH PLANNING FOR VIRTUAL ENVIRONMENTS

submitted by **TUĞCEM ORAL** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Prof. Dr. Faruk Polat
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Dept. of Computer Engineering, METU ________________

Prof. Dr. Faruk Polat
Dept. of Computer Engineering, METU ________________

Prof. Dr. Göktürk Üçoluk
Dept. of Computer Engineering, METU ________________

Asst. Prof. Dr. Selim Temizer
Dept. of Computer Engineering, METU ________________

Asst. Prof. Dr. Mehmet Tan
Dept. of Computer Engineering, TOBB ETU ________________

Date: ________________

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: TUĞCEM ORAL

Signature :

ABSTRACT

MULTI-OBJECTIVE PATH PLANNING FOR VIRTUAL ENVIRONMENTS

Oral, Tuğcem
M.S., Department of Computer Engineering
Supervisor : Prof. Dr. Faruk Polat

September 2012, 50 pages

Path planning is a crucial issue for virtual environments where autonomous agents try to navigate from a specific location to a desired one. There are several algorithms developed for path planning, but several domain requirements make engineering of these algorithms difficult. In complex environments, considering single objective for searching and finding optimal or sub-optimal paths becomes insufficient. Thus, multi objective cases are distinguished and more complicated algorithms to be employed is required. It can be seen that more realistic and robust results can be obtained with these algorithms because they expand solution perspective into more than one criteria. Today, they are used in various games and simulation applications.

On the other hand, most of these algorithms are off-line and delimitate interactive behaviours and dynamics of real world into a stationary virtuality. This situation reduces the solution quality and boundaries. Hence, the necessity of solutions where multi objectivity is considered in a dynamic environment is obvious. With this motivation, in this work, a novel multi objective incremental algorithm, MOD* Lite, is proposed. It is based on a known complete incremental search algorithm, D* Lite.

Solution quality and execution time requirements of MOD* Lite are compared with existing complete multi objective off-line search algorithm, MOA*, and better results are obtained.

Keywords: Path Planning, Multi Objectivity, Dynamic Environment

ÖZ

SANAL ORTAMLAR İÇİN ÇOK HEDEFLİ YOL PLANLAMA

Oral, Tuğcem
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Faruk Polat

Eylül 2012, 50 sayfa

Yol planlama, belirli elemanların (etmen veya robotların) tanımlanmış bir noktadan istenilen başka bir noktaya ulaşmasının söz konusu olduğu ortamlar için çok kritik bir problem oluşturmaktadır. Bilgisayar bilimleri ve robotbilim tarafından bu problemin çözümü için birçok algoritma ortaya atılmıştır. Fakat arama ve yönlendirme problemlerinin gerçek dünya uygulamalarına uyarlanması istendiğinde daha karmaşık durumlar ortaya çıkmaktadır. Bu tür karmaşık ortamlarda, en optimal veya optimale yakın yolun bulunması için tek bir kriterin göz önünde bulundurulması yetersiz kalmaktadır. Bunun bir sonucu olarak birden fazla kriteri göz önünde bulunduran yeni algoritmalar üzerinde çalışmalar yürütülmüştür. Bu algoritmaların, çözüm açısını birden fazla kriter üzerinde genişlettiği için daha gerçekçi ve daha sağlam sonuçlar ortaya koydukları gözlemlenmiştir. Günümüzde bu tür algoritmalar, oyun ve simulasyon uygulamalarında sıklıkla kullanılmaktadır.

Diğer yandan, önerilen algoritmaların çoğu çevrimdışı çalışan algoritmalardır ve gerçek dünyanın dinamikliğini ve interaktif davranışlarını durağan, sabit bir sanallıkla sınırlandırmaktadır. Bu durum çözüm sınırlarını ve kalitesini düşürmektedir. Dolayısıyla dinamik ortamlar için birden çok kriteri göz önünde bulunduran çözümlere olan ihtiyaç aşikardır. Bu güdülenme çerçevesinde; birden çok kriteri göz önünde bulundurabilen, yeni bir artımlı yol arama ve planlama algoritması, MOD* Lite ortaya çıkarılmıştır. Bu algoritma, bilinen ve bütün olan D* Lite algoritması temel alınarak geliştirilmiştir. Bulunan sonuçların kalitesi ve hesaplama süresi, çok kriterli çevrimdışı çalışan MOA* algoritmasıyla karşılaştırılmış ve çok daha iyi sonuçlar elde edilmiştir.

Anahtar Kelimeler: Yol Planlama, Çok Kriterlilik, Dinamik Ortamlar

To My Dearest Family

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my supervisor, Prof. Dr. Faruk Polat for his guidance, advice, patience and priceless insight throughout my research.

I cannot truly exaggerate my gratitude and love for my parents for their support, encouragement and love.

Also I would like to thank to all my dearest friends and especially Burçin Kelez for their encouragement, support and bringing new ideas to my perspective.

Lastly, I would like to thank to Türkiye Bilimsel ve Teknolojik Araştırma Kurumu (TÜBİTAK) / The Scientific and Technological Research Council of Turkey, for funding me during my studies.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

The problem of finding a path for an autonomous agent from an initial location to a destination location is a popular problem in real-world applications including robotics, virtual simulations or computer games and has been studied for many years. Thus, many solutions are proposed in computer science society for this issue. Proposed path planning algorithms can be classified into four categories: off-line algorithms [6] [10], on-line algorithms [13], incremental algorithms [20], [11], [19] and soft computing algorithms [25], [17]. Off-line path planning algorithms try to find the whole solution before starting the execution, whereas on-line search algorithms require the planning and execution phases to be coupled, such that the agent repeatedly plans and executes the next movement. In dynamic or partially known environments, off-line path planning algorithms suffer from execution time, whereas on-line algorithms yield low quality solutions in terms of path length. Incremental heuristic search algorithms try to merge advantages of both approaches to obtain better execution time without sacrificing optimality. They reuse the information gained from previous iterations and improve it instead of calculating from scratch like off-line search methods. Soft computing algorithms generally come up with evolutionary solutions. Their main perspective is to evaluate and evolve solution quality by time.

Existing incremental algorithms for path planning problem attempt to minimize path length. However, in many real-world problem domains we see that there are several objectives to be optimized concerning the solution (path) quality. Consider the navigation of an unmanned vehicle from one coordinate to another on a 3D terrain in a warfare setting. The navigation task is defined to be finding a path which is shortest but also the safest among all possibilities considering the existence of opponent forces in a partially known environment due to limited sensor capabilities. Note that shortest path may not be the safest one, on the contrary it might be the most dangerous one. And also the safest path may be the longest one which is unacceptable due to fuel consumption or time thresholds.

There is a need to generalize the notion of quality of a path to meet specific requirements of complex application domains where several objectives (criteria) that cannot be transformed to each other exist. For example, in our unmanned vehicle example, it is not possible to transform the distance metric to the safety metric, and vice versa. This requirement raises the problem of handling decision making of multiple criteria at the same time. In this study; an incremental path finding algorithm called Multi-Objective D* Lite (MOD* Lite) which extends an existing incremental algorithm, Dynamic A* Lite (D* Lite) [11], is introduced. MOD* Lite [16] can be used in the design of an autonomous mobile agent facing with the problem of navigation in a partially known environment that needs to optimize a predefined set of independent objectives (criteria). The agent might have limited sensor capability and hence partially observe the environment, and furthermore need to optimize multiple objectives at the same time.

In order to show that MOD* Lite generates the optimal and sub-optimal solutions, also a new multi objective genetic path planning (MOGPP) algorithm is designed. This algorithm finds initial paths randomly and its population (solutions) evolve according to a fitness function. MOD* Lite is both compared against MOGPP and MOA* algorithm [21], an offline algorithm, on some test environments that are fully observable. The performance of MOD* Lite is also tested on several partially observable environments guaranteeing the optimal solutions but outperforming the MOA* and MOGPP versions modified for unknown environments.

The following chapters in this thesis are organized as follows: Chapter 2 gives the background and related work for this study. As MOA* is used in experimental studies and D* Lite is used as a base of proposed solution, these algorithms are also detailed in this chapter. The problem definition, characteristics of the environment and proposed solutions (MOD* Lite and MOGPP) are presented in Chapter 3. Experimental studies and their results are stated in Chapter 4. Finally, the conclusion and future studies are provided in Chapter 5.

CHAPTER 2

RELATED WORK & BACKGROUND

In the literature, there are several algorithms focusing on path planning. In this chapter, existing research are introduced relevant to the context of this study where the motivation is to handle existence of multiple objectives and partial observability in path planning.

Bayili and Polat introduced a multi-objective path planning algorithm, Limited Damage A* [1] considering damage as a feasibility criterion in addition to distance. When an agent navigates in a threat zone, it is exposed to an additive damage. An upper bound is predefined for maximum damage that can be exposed and the algorithm discontinues the search on paths with damage score exceeding this threshold. The algorithm was shown to find suboptimal solutions with a reasonable time performance compared to MOA*.

Tarapata presented multi-objective approaches to shortest path problems in his study [25]. He gave a classification of multi-objective shortest path (MOSP) problems and discussed different models of them. Also he presented methods of solving the formulated optimization problems. Analysis of the complexity of the presented methods and ways of adapting of classical algorithms for solving MOSP problems were described in detail. The comparison of the effectiveness of solving selected MOSP problems were defined as mathematical programming problems and multi-weighted graph problems. Experimental results of using the presented methods for multi-criteria path selection in a terrain-based grid network were given.

Guo et al. concentrated on the problem of multi-objective path planning (MOPP) for the ball and plate system in their study [9]. The goal of MOPP was to obtain the safe -without colliding with hazardous obstacles- and shortest path for the ball to follow. The environment was represented by distance and hazard map which represents possible collisions between the ball and the obstacles. They used an entropy-based method to calculate weights of objectives for each grid node. In simulation results, the path obtained by multi-objective method was much safer when compared to single-objective A* algorithm.

In [14], Mitchell et al. examined the problem of planning a path through a low dimensional continuous state space subject to upper bounds on several additive cost metrics. For the single cost case, their previously published research has proposed constructing the paths by gradient descent on a local minimal free value function. This value function was the solution of the Eikonal partial differential equation, and efficient algorithms have been designed to compute it. In their paper, they proposed an auxiliary partial differential equation with which they evaluated multiple additive cost metrics for paths which are generated by value functions; solving this auxiliary equation adds little more work to the value function computation. They also proposed an algorithm which generates paths whose costs lie on the Pareto optimal surface for each possible destination locations, and a path can be chosen from those paths which satisfy the constraints. The procedure was practical when the sum of the state space dimension and the number of cost metrics is roughly six or less.

Evolutionary methods were also proposed for multi-objective path planning. A recent study by Pangilinan et al. [17] has introduced an evolutionary algorithm for multi-objective shortest path problem. They draw the picture of their 2-D static (stable obstacles and target) environment as a graph. Initial population was created by randomly generated individuals where each has a random ordered path from initial position to goal position. They used binary tournament selection for mating. Strength Pareto Evolutionary Algorithm (SPEA2) [26] was used to evaluate fitness values of individuals and to select them for survival. They defined density function of fitness evaluation to avoid from genetic drift. For genetic operators, they used one-point crossover and mutation. Their results show that their algorithm is a good alternative in finding a subset of efficient solutions for multi-objective shortest path problems when performance issues like complexity, diversity and non-dominal optimal solutions become obstructions.

Castillo et al. also worked on evolutionary algorithms for MOPP in their study [3]. They defined a genetic off-line point-to-point agent path planner which tries to find valid paths. They concentrated on two constraints which are path length and difficulty (each path has a difficulty which is calculated from predefined weights) in their 2-D static grid environment. They compared their results with researches from 90's and obtain better results.

Bukhari et al. came up with an optimization technique for dynamic online path planning and optimization of the path [2]. It addresses the issues involved during path planning in dynamic and unknown environments cluttered with obstacles and objects. A simulated ant agent system is proposed using modified ant colony optimization algorithm for dealing with online path planning. It is compared with evolutionary techniques on randomly generated environments; with constraints like different obstacle ratio and grid sizes. The proposed algorithm generates and optimizes paths in complex and large environments with several constraints.

Nasrollahy et al. proposed a particle swarm optimization algorithm as a multi-agent search technique, for path planning in dynamic and known (fully observable) environments in order to minimize total path planning time while avoiding local optimums [15]. They created a small-scale model of search system moving goal position and obstacles. These obstacles were defined as circular shapes and agents get around of these obstacles. They tried to optimize global best path through the goal position. Although they mentioned about effectivity of proposed algorithm, they did not give concrete results and comparisons with other methods.

Dozier et al. gave a new selection method for multi objective path planning (MOPP) in [7]. They introduced fuzzy tournament selection algorithm which combines fuzzy inference with tournament selection to select candidate solution paths. This selection was based on the euclidean distance from initial to goal position, the sum of the changes and the average change in the slope of a path.

Complete discussion of multi-objective evolutionary algorithms (MOEA) can be found in [5]. Also [4], gives a summary of current approaches in MOEA and emphasizes the importance of new approaches in exploiting the capabilities of evolutionary algorithms in multi-objective optimization.

Algorithms on incremental search aim to generate an initial sub-optimal path, and try to improve it during the consequent iterations to make it closer to the optimal. Stentz et al. proposed the Dynamic A*, D* [20] which guarantees to be optimal and is functionally equivalent to re-planning from scratch. Later, D* Lite was proposed by Koenig et al. [11] which utilized the same navigation strategy with D* but algorithmically different. It was based on Lifelong Planning A* (LPA*) [12]. D* Lite basically works as A* in the first iteration, then only updates for changed weights in environment. They prove that D* Lite was at least as efficient as D*.

2.1 Multi Objective A* (MOA*)

Classical A* [10] is a complete and optimal solution for the cases where only a single optimization criteria is crucial for path cost. On the other hand, real-world applications generally consider more than one criteria at the same time, which could not be converted, reduced or combined with each other. In this manner, multi objective A* (MOA*) [21] extends classical A* to handle multiple objectives that inherently exist in many application domains. It uses the evaluation function $F(n) = G(n) + H(n)$ similar to A* but functions return vectors instead of scalar values. Size of the vector is the number of objectives to be optimized. If there is only one objective MOA* becomes standard A*. Like A*, it provides complete and optimal solutions when heuristic function is admissible which means the heuristic estimation of every objective is not overestimated.

MOA* keeps track of state expansions using *OPEN* (to be processed nodes) and *CLOSED* (already processed nodes) sets. Non-dominated states are maintained in a subset of *OPEN* named *ND* which is formed by the elements that are not dominated by any other element of this set and any of the discovered solutions. The overview of MOA* is given in Figures 2.1 and 2.2.

At each iteration of the algorithm, first the best alternative node is selected from *ND*. Then, the selected node is checked whether it is in the set of goal nodes or not. If so, the current node and its path cost vector are added to the solution set and the iteration continues with selection of a new node. Otherwise, the adjacent nodes of current node are generated. At this step, for all generated adjacent nodes, the newly

0. Initialize by setting $OPEN$ equal to a set containing only the start node and setting $CLOSED$, $SOLUTION$, $SOLUTION\_COSTS$, $SOLUTION\_GOALS$, and $LABEL$ each equal to the empty set.
1. Find the set of nodes in $OPEN$, call it ND, that have at least one node selection function value that is not dominated by:
    1.1. the cost of any solution path already discovered (i.e., in $SOLUTION\_COSTS$), nor by
    1.2. the node selection function values of any other potential solution represented by a node on $OPEN$.
2. Terminate or select a node for expansion.
    2.1. If ND is empty, do the following:
        2.1.1. Use the set of preferred solution path costs in $SOLUTION\_COSTS$ and the $LABEL$ sets. If any, to trace through backpointers from the goal nodes in $SOLUTION\_GOALS$ to s.
        2.1.2. Place any solution paths in $SOLUTION$.
        2.1.3. Stop.
    2.2. Otherwise, do the following:
        2.2.1. Use a domain-specific heuristic to choose a node n from ND for expansion, taking goals, if any, first.
        2.2.2. Remove n from $OPEN$.
        2.2.3. Place n on $CLOSED$.
3. Do bookkeeping to maintain accrued costs and node selection function values.
4. Identify solutions.
    4.1. If n is a goal node, do the following:
        4.1.1. Add it to $SOLUTION\_GOALS$,
        4.1.2. Add its current costs to $SOLUTION\_COSTS$.
        4.1.3. Remove any dominated members of $SOLUTION\_COSTS$.
        4.1.4. Go to Step (6).
    4.2. Otherwise, continue.

Figure 2.1: Multi Objective A* Outline

5. Expand n and examine its successors.

   5.1. Generate the successors of n.
   5.2. If n has no successors, go to Step (6).
   5.3. Otherwise, for all successors n' of n, do the following:

      5.3.1 If n' is a newly generated node, do the following:

         5.3.1.1 Establish a backpointer from n' to n.

            5.3.1.1.1 Set $LABEL(n', n)$ equal to the nondominated subset of the set of accrued costs of paths through n to n' that have been discovered so far.

         5.3.1.2 Establish a nondominated accrued cost set, $G(n') = LABEL(n', n)$,

         5.3.1.3 Compute node selection values, $F(n')$, using $G(n')$ and $H(n')$.

         5.3.1.4 Add n' to $OPEN$.

      5.3.2 Otherwise, n' was previously generated. so do the following:

         5.3.2.1 If any potentially nondominated paths to n' have been discovered; then, for each one, do the following:

            5.3.2.1.1 Ensure that its cost is m $LABEL(n', n)$ and therefore in the current set of nondominated accrued costs of paths discovered so far to n'; that is, in $G(n')$.

            5.3.2.1.2 If a new cost was added to $G(n')$, do the following:

               5.3.2.1.2.1 Purge from $LABEL(n', n)$ those costs associated with paths to n' to which the new path is strictly preferred.

               5.3.2.1.2.2 If n' was on $CLOSED$, move it to $OPEN$.

6. Iterate.

   6.1. Increment iteration counter.
   6.2. Go to Step (1).

Figure 2.2: Cont'd of Multi Objective A* Outline

generated node n' is checked for being generated for the first time. If so, its path cost estimate vector $F(n')$, traversed path cost vector $G(n')$ and the heuristic estimate vector $H(n')$ are computed, and the newly generated node is added to *OPEN* set. If the node is not explored for the first time, there is a possibility that a path passes through this node with non-dominated costs to other candidates. Then the node and its non-dominated cost vectors are taken into consideration in the following steps of the solution. The algorithm iterates over the above steps until the *ND* set becomes empty. Finally, solution paths are generated by following back-pointers from goal to start.

2.2 D* Lite

D* Lite [11] is one the of most popular goal-directed autonomous agent navigation algorithms and widely used in unknown environment. It is an adaptation of Lifelong Planning A* [12] which is an incremental derivation of A* [10]. It determines the same paths as D* Algorithm [20] and moves the agent the same way but it is algorithmically different. Incremental search methods reuse information from previous searches to find solutions to similar problems much faster than is possible by solving each search task from scratch.

D* Lite Algorithm is a reverse or backward searching method where searching starts from the target position. It is able to re-plan from current position when a weight has been changed in the environment, i.e. there is a new obstacle blocking the path. It divides the environment into grids and path finding and agent's movement are from grid to grid.

2.2.1 Overview

Before considering an introduction to the problem that D* Lite covers, think about an agent path planning and navigation task in a dynamic unknown environment. In this scenario, agent always observes its current cell's neighbour cells and try to move one of them if it is traversable. Agent starts from start cell and moves through to the goal cell. It always try to compute the shortest (or minimized some other cost metric as determined by edge cost) path assuming that unexplored cells are traversable. Next, the agent try to follow found path until an untraversable cell is observed or reached to target cell successfully. Otherwise, the agent should recompute a shortest path from its current location to the target. Figure 2.3, simply represents the knowledge of cell states before and after of a movement. Each cell shows the goal distances from the agent's current cell to the goal. Known shortest path is drawn by an arrowed line. All of the cells in the grid except the adjacent of start cell are unexplored before the agent has moved and are assumed to be traversable; these cells are painted white. Cells are shaded gray in the lower grid maze whose goal distances have changed during discovery. The efficiency of D* Lite comes from replanning path just according to these changed cells.

2.2.2 The Algorithm

This subsection represents the notation, gives details of used core components and elaborates every single step of D* Lite.

2.2.2.1 Notation & Formulation

The notation is defined for D* Lite as follows: Let S denotes the finite set of vertices of the graph. Let $Succ(s) \subseteq S$ and $Pred(s) \subseteq S$ denote the set of successors and predecessors of vertex $s \in S$, respectively.

The cost of moving from vertex s to $s' \in Succ(s)$ is denoted by $0 < c(s, s') \leq \infty$. D* Lite always gives shortest path found between s_{start} and s_{goal} where $s_{start}, s_{goal} \in S$. $g^*(s)$ is used to define the distance from s_{start} to a vertex s. Heuristic function of D* Lite is also similar with classic A*, where $h(s_{goal}, s_{goal}) = 0$ and $h(s, s_{goal}) \leq c(s, s') + h(s', s_{goal})$; triangular inequality is held for all vertices $s \in S$ and $s' \in Succ(s)$ with $s \neq s_{goal}$.

D* Lite maintains an estimate $g(s)$ of the start distance $g^*(s)$ of each vertex s, analogous to the *g-values* of an A* search. D* Lite carries them forward from search to search. D* Lite also maintains a second kind of estimate of the start distances; the *rhs-values* are one step lookahead values based on the *g-values* and thus potentially

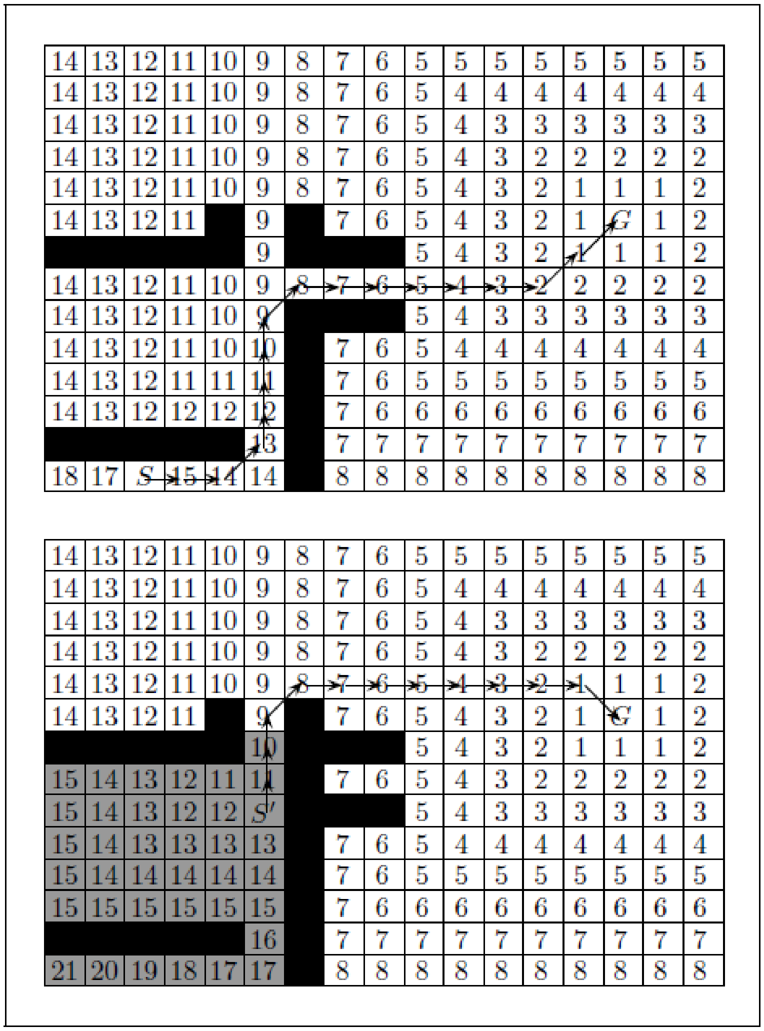


Figure 2.3: A Simple Grid Search Representation

better informed than the *g-values*. The *rhs-values* should always satisfy the following equation;

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start}; \\ min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

A vertex is stated as locally consistent if and only if its *g-value* is equal to its *rhs-value*, otherwise it is locally inconsistent. In the case that all vertices are locally consistent, the *g-values* of all vertices are equal to their start distances, $g^*(s)$ namely. Actually, D* Lite does not try to make all vertices locally consistent after some edge costs have changed. It is not required to recompute start distances which have been computed before and have not been changed. Also, it uses admissible heuristic information in order to focus the planning phase and updates only the related *g-values* which are relevant to the computation of a shortest path.

The D* Lite algorithm is complete; it always finds a shortest path if one exists or terminates. If $g(s_{goal}) = \infty$ after the search, then no path is constructed between s_{start} and s_{goal}. Otherwise, one can trace a shortest path from s_{start} to any vertex s_u by, starting at vertex s_u, and always tracing back from the current vertex s to any predecessor s' of s that minimizes $g(s') + c(s', s)$ until s_{start} is reached. Notice that ties are broken arbitrarily.

2.2.2.2 Core Components and Details

Consistency of a vertex in the search graph could be considered in several conditions. A vertex s is called *locally consistent* iff $g(s) = rhs(s)$ and *locally inconsistent* iff $g(s) \neq rhs(s)$. Local inconsistency refers to those whose *g-values* need to be updated to become locally consistent. In the same manner, a locally inconsistent vertex is called *locally overconsistent* iff $g(s) > rhs(s)$ and *locally underconsistent* iff $g(s) < rhs(s)$. These consistency situations are used to manage vertices.

Like A*, D* Lite also maintains a priority queue with heuristic information such that the most promising vertices are expanded first. This queue contains only the locally inconsistent vertices which are selected sequentially to be expanded according to their key values $k(s)$, a vector of two components:

$$k(s) = [k_1(s); k_2(s)]$$

$$k_1(s) = min(g(s), rhs(s) + h(s, s_{goal}))$$

$$k_2(s) = min(g(s), rhs(s))$$

The first component of the key $k_1(s)$ corresponds to $f(s) = g*(s) + h(s, s_{goal})$ which is the *f-value* of A*, and the second component $k_2(s)$ corresponds to the *g-value* of A*. Keys are compared (and maintained in the priority queue) in lexicographic order where $k(s) \leq k(s')$ iff either $k_1(s) < k_1(s')$ or $k_1(s) = k_1(s')$ and $k_2(s) \leq k_2(s')$.

Thus, the key with the smallest value is taken from the priority queue and expanded. The queue has several functionalities; where *top()* returns the vertex with the smallest priority, *topKey()* returns the key value of the vertex at the top or $[\infty, \infty]$ if the queue is empty. *pop()* removes and returns the vertex with the smallest key and finally, the member functions *remove()* and *insert()* removes a vertex from and inserts a vertex into the queue, respectively.

The D* Lite algorithm is started by calling *plan()* method in Algorithm 2. It first calls *initialize()* in Algorithm 1 at {3} to start the search. *initialize()* sets priority queue U to an empty set, the heap reordering variable k_m to zero, and the g and *rhs-values* of all vertices to infinity. Initially, s_{start} is the only consistent vertex and is inserted into the empty priority queue with its calculated key $k(s_{start})$. *calculateKey()* method is used to calculate the key of corresponding given vertex. Key formulation and generation is indicated above. This initialization guarantees that the first call to *computeShortestPath()* at line {4} in Algorithm 2 performs an A* search that it expands the same vertices as A* would, in exactly the same order.

After first execution of *computeShortestPath()*, a loop is processed until reaching to s_{goal}. In each iteration, when $g(s_{goal})$ is observed as ∞, one can infer that there is no known path between s_{start} and s_{goal}. If this is the case, the algorithm can be terminated. Else, next vertex to move is determined as new s_{start} with the equation in line {7} in Algorithm 2 where minimum cost adjacent of current s_{start}.

At this point, a change in the edge costs is waited in the environment at line {9}. If any edge costs have changed, the heap variable k_m is cumulatively updated with

Algorithm 1 D* Lite Outline

function CALCULATEKEY(s)
  return $[min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))]$;

function INITIALIZE()
  $U = \varnothing$;
  $k_m = 0$;
  for all $s \in S$ **do**
    $rhs(s) = g(s) = \infty$;
  $rhs(s_{goal}) = 0$;
  $U.insert(s_{goal}, calculateKey(s_{goal}))$;

function UPDATEVERTEX(u)
  if $u \neq s_{goal}$ **then**
    $rhs(u) = min_{s' \in succ(u)}(c(u, s') + g(s'))$;
  if $u \in U$ **then** U.remove(u);
  if $g(u) \neq rhs(u)$ **then**
    U.insert(u, calculateKey(u));

function COMPUTESHORTESTPATH()
  while $U.topKey() < calculateKey(s_{start}) \mid\mid rhs(s_{start}) \neq g(s_{start})$ **do**
    $k_{old} = U.topKey()$;
    $u = U.pop()$;
    if $k_{old} < calculateKey(u)$ **then**
      U.insert(u, calculateKey(u));
    else if $g(u) > rhs(u)$ **then**
      $g(u) = rhs(u)$;
      for all $s \in pred(u)$ **do** updateVertex(s)
    else
      $g(u) = \infty$;
      for all $s \in pred(u) \cup u$ **do** updateVertex(s);

Algorithm 2 Cont'd of D* Lite Outline

function PLAN()
    $s_{last} = s_{start}$
    initialize();
    computeShortestPath();
    while $s_{start} \neq s_{goal}$ **do**
        if $g(s_{start}) = \infty$ **then** there is no known path
        $s_{start} = argmin_{s' \in succ(s_{start})}(c(s_{start}, s') + g(s'))$;
        Move to s_{start};
        Scan the graph for changed edge costs;
        if Any weight cost changed **then**
            $k_m = k_m + h(s_{last}, s_{start}))$;
            $s_{last} = s_{start}$;
            for all directed edges (u,v) with changed edge costs **do**
                Update the edge cost c(u,v);
                updateVertex(u);
            computeShortestPath();

heuristic function of last visited and s_{start} vertices. Then; for all changed edge costs, new cost $c(s, s')$ is recalculated and *updateVertex()* is called to recompute the *rhs-values* and keys of the vertices potentially affected by the changed edge costs. In addition, if any of the vertices potentially affected have become locally consistent or inconsistent, their membership in the priority queue is adjusted. The k_m variable is important because repeated reordering of priority queue U is expensive since it often contains large number of vertices. By cumulatively adding heuristic value between last visited vertex and start, whenever new priorities are computed, the variable k_m has to be added to key value's first components. In this way, the order of vertices in the priority queue is unaffected when the agent moves and the priority queue doesn't need to be reordered.

Finally, *computeShortestPath()* is called which repeatedly expands locally inconsistent vertices according to their priorities. If top key of U, k_{old} is smaller than calculated new key of corresponding vertex u, this vertex is inserted into queue. When it expands a locally overconsistent vertex u at {22} in Algorithm 1, *g-value* is set as *rhs-value* to make u locally consistent. When a locally underconsistent vertex u is expanded in {25}, the *g-value* of u is set to infinity. This makes the corresponding vertex u either locally consistent or overconsistent. If it was locally overconsistent, then changing of its *g-value* can effect the local consistency of its neighbours, or successors. Otherwise, if u was locally underconsistent, then changing of its *g-value* can effect the local consistency of itself and its neighbours. As a result, *computeShortestPath()* must *updateVertex()* for all of the vertices potentially effected by the change in their *g-values*. It modifies their *rhs-values*, checking their consistency, and adding them to or removing them from the priority queue as appropriate in *updateVertex()* method in lines $\{10 - 15\}$. The vertices are expanded by *computeShortestPath()* until the key of the next vertex s' to be expanded is no less than that of s_{goal} or until s_{goal} is locally consistent. This behaviour is similar with A* where expands vertices until it expands s_{goal} at which point the *g-value* of s_{goal} is equal to its start distance and the *f-value* of the node to expand next is no less than the *f-value* of s_{goal}.

At the end of the algorithm execution, one can trace back a shortest path from s_{start} to s_{goal} by always transitioning from the current vertex s, starting at s_{goal} , to any predecessor s' that minimizes $g(s') + c(s', s)$, breaking ties arbitrarily, until s_{start} is reached.

CHAPTER 3

PROPOSED SOLUTIONS

3.1 Motivation

Assume that an unmanned aerial vehicle (UAV) is taking off from an initial location. Its goal is trying to shoot an enemy unit on a predefined target location in an unknown dynamic environment. However, this enemy unit is protected by air defence units scattered on the terrain having different capabilities (hit ratios) and coverage areas. Each defence unit scans the space within its coverage areas to detect any threat. Due to its limited sensor capability, a UAV can only partially observe the environment. The air defence zones produce computable risk values for UAVs when they enter UAV's perceived sensor range. On the other hand; UAV has limited fuel and time, so it must locate and shoot the target quickly but the risk of being hit by an air defence unit must be minimized. This means that the UAV should find both shortest and safest path *as quickly as possible.*

In this real-world problem, the UAV has to execute a planner and quickly find available paths. Also it must re-plan executed path when an unknown part of the environment become known or known parts are changed (i.e. a visible defence unit's coverage area changes, shrinks or enlarges; or defence unit is disabled / enabled) as it navigates. Mostly, evolutionary search algorithms focus on this issue and come up with several solutions like [18], [8]. However, It is obvious that these algorithms are insufficient for reflecting and adapting the dynamics of the environment as they are not incremental. One alternative could be to adapt off-line MOA* to unknown environments but it is grossly inefficient as it has to be restarted from scratch every time when some unknown part of the environment becomes known or known part changes.

Considering these issues within path planning perspective, a necessity of a solution can be observed clearly. Thus, a multi objective incremental path planning algorithm is designed and developed based on non-optimized version of D* lite [11] in this study. This algorithm is called multi objective D* lite, or MOD* Lite [16]. To prove effectiveness of MOD* Lite, an alternative solution based on a genetic algorithm, multi objective genetic path planner, or MOGPP is also developed. Both solutions, MOD* Lite and MOGPP are detailed in next sections.

3.2 Multi Objective D* Lite : MOD* Lite

3.2.1 Overview

Multi-objective problems focus on considering more than one objective concurrently (at the same time). Consequently, *all* scalar values and atomic operations (like addition, checking for equality, etc.) are to be converted into vectors of scalars. This causes all functions (cost, heuristic, etc.) to have n dimensions if there are n non-interacting objectives to be optimized. For two scalars a and b, there are three outcomes: $a < b$, $a > b$ or $a = b$. However; for two vectors u and v, besides $u < v$, $u > v$ and $u = v$ there is a fourth alternative meaning that u and v cannot be compared. u and v are said to be *equal* if all corresponding objective values are equal, or in other words;

$$\forall n \in N; u_n = v_n$$

where N is total number of objective values for u and v, n is the n^{th} objective value. For equality, both u and v must have the same cardinality, in other words, the same amount of objectives.

It could be said that u *dominates* v if u is better in at least one objective compared to v or in other words, there is no objective of v where it is better than any objective of u. Moreover, u and v are said to be *non-dominated* if for at least one objective, u is better but for at least some other objective, v is better. For instance, assume that we have two objectives to be minimized and let $v_1 = [3, 4], v_2 = [4, 6], v_3 = [6, 2]$. Here v_1 dominates v_2 but v_1 and v_3 are non-dominated.

MOD* Lite enables a user to define a set of objectives, $O_1, O_2, \cdots, O_n$ to be used in the evaluation of the quality of the candidate paths explored by an incremental algorithm. In that respect MOD* Lite is a domain-independent path search algorithm that can be used in any search problem where the environment is partially or fully observable. Note that for each objective O_i, the user needs to define whether O_i is to be minimized or maximized. Each objective is assumed *not to be transferable* to and *non-interacting* with each other. This situation might reduce the number of objectives and should be considered out of this study, with a different algorithm. For the sake of simplicity, the execution and tests of MOD* Lite is restricted to the UAV domain example which has been introduced above with two objectives to be minimized, namely the distance and the degree of risk of danger.

3.2.2 Environmental Properties

MOD* Lite applied to the UAV path finding task is illustrated in a 2-D grid based environment. It is easier to present the algorithm and also demonstrate its effectiveness on such a simple environment.

The environment is considered to be partially observable because of limited sensor capability of the agent. The agent can perceive the environment around her within a square region centered at the agent location. The size of the square is $(2v + 1)$ x $(2v + 1)$, where v is the vision range. As the agent navigates, the known part of the environment gradually increases and it is presumed that agent is designed as having enough memory space to maintain all the perceived environment. It is assumed that the target is stationary and its location is known by the UAV agent at the initial step. Furthermore, the environment has randomly placed obstacles that cannot be traversed by the agent. The agent occupies only one grid cell. There are also threat zones in the environment. Threat zones produce predefined risk values which could effect the agent to fail reaching to the target cell. Threat zones are constructed up to three sub-zones. The innermost one is more hazardous than outer ones. So, if the agent has to enter a threat zone, it prefers to pass through outer levels. With threat zones, the agent must think about both the shortest and the safest path. The environment has randomly placed different sized threat zones.

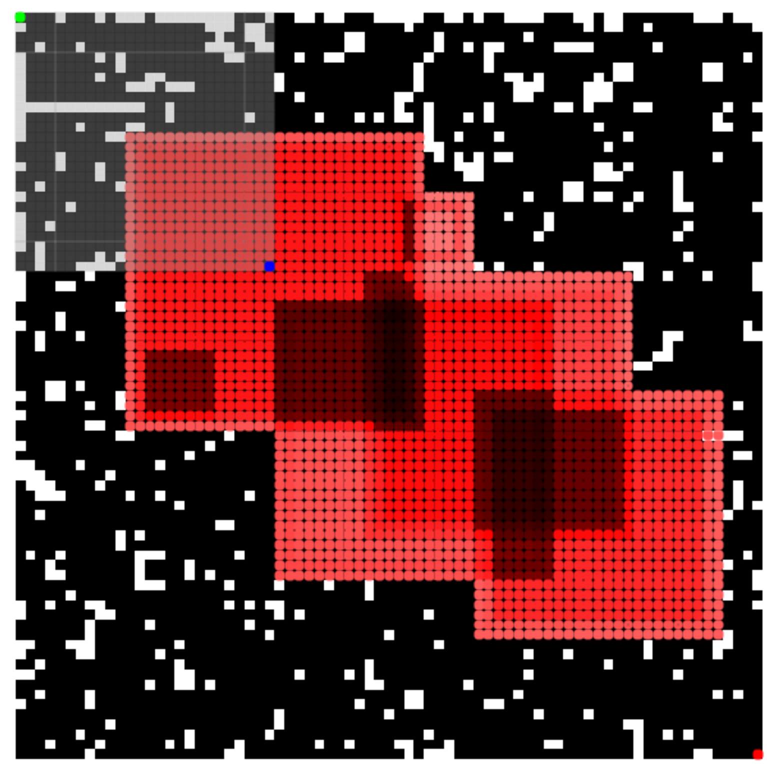


Figure 3.1: A 75 x 75 Visualized Grid Environment

A simple representation of 2-D grid environment is given in Figure 3.1. In this figure, white cells are obstacles and black cells are available ones. Agent can traverse all cells except for white ones. The UAV agent is initially placed on upper leftmost cell and painted as green. The target locates on the bottom rightmost and marked as dark red color. The lighter-red square areas represent threat zones. As two threat zones overlaps, the overlapped area becomes darker to represent that risk is increased exponentially on this area. The fogged gray area around UAV agent represents agent's vision range. As agent moves, this area is updated. Thus, agent always chooses the best available cell within her vision range with respect to actual target. This temporary target is marked with blue.

3.2.3 The Components and Variables

MOD* Lite is the multi-objective extension of D* Lite. It can be applied to any unknown dynamic multi-objective search problem where costs can change by time.

Considering formal definition; S denotes the set of states in search problem. $s_{start} \in S$ and $s_{goal} \in S$ are the initial and final (target) states, respectively. $pred(s) \subseteq S$ and $succ(s) \subseteq S$ can be used to find predecessors and successors of given state, s. The heuristic function $h(s, s')$ which estimates costs between s and s', cost function $c(s, s')$ which represents the actual cost traversing from s to s', actual cost function $g(s)$ and the $rhs(s)$, one-step-lookahead values of $g(s)$ functions are all inherited from D* Lite. However, as we have to consider more than one objective, these functions are to return vector values of scalars instead of scalars. Thus, $rhs(s)$ satisfies the condition

$$rhs(s) = \begin{cases} ObjectiveVector.MIN & \text{if } s = s_{start}; \\ nonDom_{s' \in pred(s)}(sum(g(s'), c(s', s))) & \text{otherwise.} \end{cases}$$

where ObjectiveVector.MIN stands for a vector with n minimum values for an n dimensional problem. These values could be 0 or ∞ for minimization or maximization of objective, respectively. $sum()$ function implements vector summation and $nonDom()$ function returns the set of best non-dominated vectors corresponding to predecessors of any state s. Note that $nonDom()$ constructs a list of objective vectors, $rhs(s)$ and $g(s)$ function values are represented as a *lists of objective vectors* where each objective vector in this list is non-dominated with others. An objective vector is a structure that holds values for each objective defined by problem (minimization or maximization). In case of having more than one objective, it is possible that there are more than one paths to a particular state that do not dominate each other. That's why each state might be represented by several vectors. In case we need to compare if one state is better than another, two sets of their vectors need to be compared as formulated below.

Definition It can be said that u *completely dominates* v iff $\forall x \in u$ and $\forall y \in v$; x dominates y.

Assume that $\upsilon_1 = \{[2,5],[3,4]\}, \upsilon_2 = \{[6,1],[5,2]\}, \upsilon_3 = \{[3,5]\}$ are lists of objective vectors for three states. υ_1 and υ_2 are non-dominated whereas υ_1 completely dominates υ_3.

It could be frankly said that the terms "greater than", "smaller than" and "equals" for scalar value comparison are replaced by "completely dominates", "completely dominated by" and "multi-objectively equals" for vectors of scalars. Non-domination is introduced and handled as the fourth case.

D* Lite introduces local consistency and inconsistency concepts with respect to comparing $g(s)$ and $rhs(s)$. A state is called *locally consistent* when $g(s)$ and $rhs(s)$ are equal, and *locally inconsistent* otherwise. A locally inconsistent state is referred as locally underconsistent if $g(s) < rhs(s)$ or locally overconsistent if $g(s) > rhs(s)$. In the case of non-domination of these functions, we introduce the concept of *local non-consistency*:

Definition A state is referred as *locally non-consistent* if its $g(s)$ and $rhs(s)$ values are non-dominated to each other. This inconsistency condition causes the state to reside on more than one solution because it can be understood that two or more predecessors of s are non-dominated to each other.

Other multi-objective operations are introduced in following subsections. The overall flow of MOD* Lite is given in Algorithm 3.

Basically, D* Lite tries to make all states locally consistent. Locally inconsistent states are maintained in a priority queue (U) with their key values and expanded considering priority values. However, locally non-consistent states cannot be maintained in such a queue due to the non-domination of their key values, which are also set of objective vectors. If two keys cannot be dominated by each other, they should be criticized in the same manner. Thus, a more convenient structure; a directed acyclic state expansion graph instead of a priority queue which uses topological ordering of states with respect to the key domination, is presented. In this model, the graph (U) contains set of nodes each represented by a state and its key value. When a state is to be added into U with $insert(state, key)$ operation, key value is compared with all existing nodes' key values. If the new state dominates some state, an edge is introduced from the new state to this state. No edge connection is done in case of multi objectively equality and non-domination. As a result, incoming and outgoing degrees of a node s correspond to the number of nodes that *dominates* s and the number of nodes that are *dominated by* s, respectively. The node(s) with incoming degree 0 are the non-dominated nodes

Algorithm 3 Main loop of MOD* Lite

function CALCULATEKEY(s)
    k_2(s)= nonDom(g(s), rhs(s))
    k_1(s) = sum(h(s_{start}, s), k_m, k_2(s))
    return $[k_1(s), k_2(s)]$

function INITIALIZE()
    $U = \varnothing$
    k_m = ObjectiveVector.MIN
    for all $s \in S$ **do**
        rhs(s)=g(s)=ObjectiveVector.MAX
    rhs(s_{goal}) = ObjectiveVector.MIN
    U.insert(s_{goal}, calculateKey(s_{goal}))

function PLAN()
    initialize()
    computeMOPaths()
    while *true* **do**
        solutionPaths = generateMOPaths()
        if solutionPaths = null **then** there is no known path
        Wait for any weight cost to change;
        if Any weight cost changes **then**
            k_m = sum(k_m, h(s_{goal}, s_{start}))
            for all Changed weight costs of edges(u,v) **do**
                Update cost c(u,v)
                updateVertex(u)
            computeMOPaths()

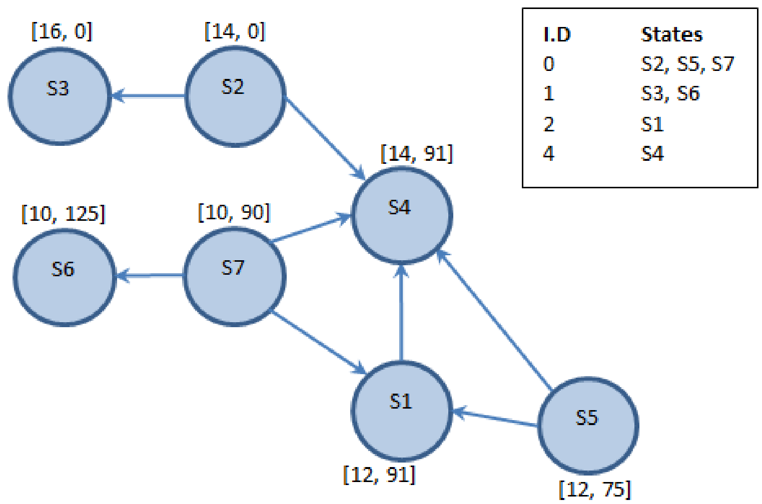


Figure 3.2: Directed Acyclic State Expansion Graph

where none of other nodes could dominate. $topKey()$ and $topKeys()$ return the key value(s) of nodes with minimum incoming degree. $pop()$ returns and removes the state (all its incident -incoming and outgoing- edges are also removed from the graph) with minimum incoming degree. Another strategy for $pop()$ operation could be removing the state with maximum outgoing degrees where the number of outgoing degree of a state s presents that how many states are dominated by s. Also, these two methods could be combined where two topologically ordered lists, one is for minimum incoming degrees and the other is for maximum outgoing degrees, are maintained and a state which has both minimum incoming and maximum outgoing degrees according to these lists could be selected. As these strategies could be set to applied domain according to its requirements, we choose selecting and removing from minimum incoming degrees list. If more than one nodes exist with minimum degree, one of them is selected randomly. $remove(state)$ operation removes a given state and its incident edges from graph. An example of a state expansion graph with states and their corresponding key values is given in Figure 3.2. Incoming degrees of nodes are given as a list in the figure.

Addition of a new state to the state expansion graph is illustrated in Figure 3.3. $S8$ is "the new" state to be added, the dashed directed edges are established between $S8 - S1$, $S8 - S4$ and $S8 - S5$ because $S8$'s key can only dominate keys of nodes $S1, S4$ and $S5$. None of the existing states' keys can dominate $S8$, so incoming degree

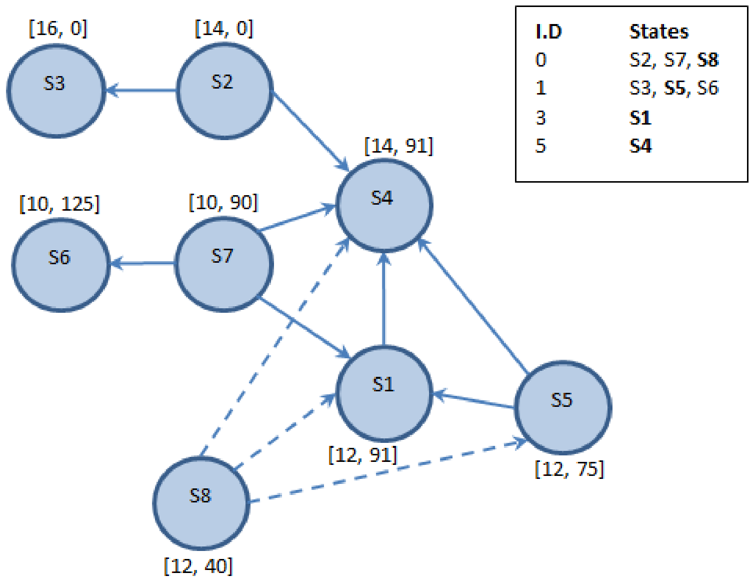


Figure 3.3: State Expansion Graph after Adding State $S8$

of $S8$ becomes 0. This addition also effects the incoming degrees list where the changed positions are highlighted in the figure. Addition of $S8$ increments the incoming degrees of $S1$, $S4$ and $S5$ by 1 so their positions are shifted down.

3.2.4 Key Formulation

In the previous subsection, it is stated that the directed acyclic graph structure is used to determine expansion of nodes in state space with their *keys*. The basic idea behind the calculation of keys is similar with D* Lite, with slight modifications. As MOD* Lite is considered in a multi-objective setting, key value is stated as a vector with two components: $k(s) = [k_1(s); k_2(s)]$ where these components are set of objective vectors. $k_2(s)$ is calculated by finding the *non-dominated list* of $g(s)$ and $rhs(s)$, where $k_2(s) = nonDom(g(s), rhs(s))$. The other component, $k_1(s)$ is calculated as vector summation of $h(s_{start}, s), k_m$ and $k_2(s)$. Calculation of $k(s)$ can be seen in lines $\{2$ and $3\}$ in Algorithm 3. k_m is used for heap reordering as defined in D* Lite.

Algorithm 4 Update Vertex & Compute Multi-Objective Paths

```
function updateVertex(u)
    if u ≠ s_goal then
        rhs(u) = nonDom_{s' ∈ succ(u)}(sum(c(u, s'), g(s')))
    if u ∈ U then U.remove(u)
    if !equals(g(u), rhs(u)) then
        U.insert(u, calculateKey(u))

function computeMOPaths()
    while dominatesAll(calculateKey(s_start), U.topKeys()) do
        k_old = U.topKey()
        u = U.pop()
        k_new = calculateKey(u)
        if k_old.completelyDominates(k_new) then
            U.insert(u, k_new)
        else if rhs(u).completelyDominates(g(u)) then
            g(u) = rhs(u)
            for all s ∈ pred(u) do updateVertex(s)
        else if g(u).completelyDominates(rhs(u)) then
            g(u) = ObjectiveVector.MAX
            for all s ∈ pred(u) ∪ {u} do updateVertex(s)
        else
            g(u) = nonDom(g(u), rhs(u))
            for all s ∈ pred(u) do updateVertex(s)
```

3.2.5 Details of MOD* Lite

MOD* Lite is based on D* Lite algorithm as introduced in the previous section. There are fundamental differences due to the structures used and the way the solution paths are maintained. The pseudocode of MOD* Lite is given in Algorithms 3, 4 and 5.

First of all, searching order of MOD* Lite is from goal to start state, like D* Lite. The main function of MOD* Lite first calls $initialize()$ to set up the execution in Algorithm 3. This function calculates the key value for goal state, adds it into U and sets the rhs value to a *MIN* objective vector, which has minimized n values for n-objectives. These values can be 0 for minimized-objective and ∞ for maximized-objective. Then, proper $g(s)$ values are calculated considering all objectives with $computeMOPaths()$. Finally, paths are generated with these $g(s)$ values. If a weight cost is changed in the environment, corresponding states are re-expanded and only related weights are updated. Notice that this cost change might happen for only one objective or several objectives at the same time.

The $computeMOPaths()$ pseudocode is given in Algorithm 4 line $\{7\}$. The termination criteria of this function is where the key of s_{start} dominates all the top keys returned from U. Until it terminates, the top state is sequentially selected from top states of U and expanded. While expanding a state, the domination between g and rhs values of corresponding state is observed. If $rhs(s)$ values completely dominate $g(s)$ values, local underconsistency case occurs. We apply the same strategy with D* Lite, update g value with rhs and update weights for all predecessors of s with $updateVertex()$. If $g(s)$ values completely dominate $rhs(s)$ values, the case is locally overconsistency. Simply g value for this state is set as *MAX* objective vector, which is ∞ for minimized-objective and 0 for maximized one, and current state weight is updated with its predecessors' weight. The third case occurs when g and rhs values can not completely dominate each other, *locally non-consistency.* In this case, g value is updated with non-dominated values of g and rhs values and again predecessors of current state is updated. Keeping non-dominated values of g and rhs enables to keep track of each non-dominated successors' information.

To update a weight of a state, MOD* Lite uses $updateVertex(u)$ shown in Algorithm

4 line {1-6}. It simply adds corresponding state to or removes from U according to given criteria. While updating $rhs(u)$ except goal state, non-dominated objective values of multi-objectively summed $c(u, s')$ and $g(s')$ are established and used.

After state expansion operation is finalized and corresponding $g(s)$ values are set, multi-objective paths are generated via these $g(s)$ values by given pseudocode in Algorithm 5. Path generation is achieved in two phases: setting parent(s) for each non-dominated successor of expanding state and constructing paths by following (backtracking) these parents. The first phase is performed from the start to the goal state whereas the second is from the goal to the start state.

A queue is used to keep track of expanding states which is shown in line {2}. This queue initially has s_{start} only. Thus, starting from s_{start}, the while loop iterates until this queue becomes empty. Finding a goal state is not considered as a termination criteria because other non-dominant paths might be available. As expanding a state, we refer to set *it* as a parent to its successors indicated between lines {7-36}.

Before expansion of a state s, non-dominated successors are found first with respect to multi-objective summation of $c(s, s')$ and $g(s')$ as shown in line {5}. If a successor s' is found in non-dominated successors list, it has a potential to have s as a parent. For each non-dominated successor s', first parents list of s is checked. If s does not have any parent, which only occurs iff $s = s_{start}$, for sure s' does not have any parent as well. In this case, s is added as a parent of s' with corresponding cost $c(s, s')$. Parents of a state are kept in a map where keys of this map are parents and values are cumulative costs which is consumed to reach that state from start through corresponding parent. These costs are used to determine elimination of existing parents when a new one is considered to be added. This idea will be elaborated later.

If s has predefined parents (starting from {9}), a cumulative total cost is calculated for s' in line {10}. This cost is multi-objective summation of $c(s, s')$ and aggregated cost values of parents of s. Notice that the algorithm proves that parents' costs of a state are always non-dominated to each other, so the aggregated cost values contain *all* parents' *all* costs. These cost values express all non-dominated solution costs to reach that state. If s' does not have any parent up to now ({11}), s is added as a parent of s' with cumulative cost. Else, each existing parent of s', say s'' should be compared

Algorithm 5 Path Generator Algorithm

```
function generateMOPaths()
    expandingStates.add(s_start)
    while !expandingStates.isEmpty() do
        s = expandingStates.poll()
        nonDomSuccs = nonDom_{s' ∈ succ(s)}(sum(c(s, s'), g(s'))
        for all s' ∈ nonDomSuccs do
            if s.parents() = null then
                s'.parents().put(s, c(s, s'))
            else
                cumulativeC = sum(c(s, s'), s.parents().values())
                if s'.parents() = null then
                    s'.parents().put(s, cumulativeC)
                else
                    for all s'' ∈ s'.parents() do
                        if equals(s'.parents(s"), cumulativeC) OR completelyDominates(
    s'.parents(s"), cumulativeC) then
                            break
                        else if completelyDominates(cumulativeC, s'.parents(s")) then
                            s'.parents().remove(s")
                            s'.parents().put(s, cumulativeC)
                        else
                            for all cC ∈ cumulativeC do
                                for all eC ∈ s'.parents(s'') do
                                    if eC.equals(cC) OR eC.dominates(cC) then
                                        cumulativeC.remove(cC)
                                        break
                                    else if cC.dominates(eC) then
                                        s'.parents(s").remove(eC)
                                        break
                                if s'.parents(s") = null then
                                    s'.parents().remove(s")
                            if !cumulativeC = null then
                                s'.parents().put(s, cumulativeC)
            if s'.parents.contains(s) AND !expandingStates.contains(s') then
                expandingStates.add(s')
    solutionPaths = construct paths recursively traversing parents
    return solutionPaths
```

with the cumulative cost. These operations are shown in lines between {13-32}. Here, if s'' has same cost with or better cost (determined by completely domination term) than cumulative cost, needless to say that s is not required to be added as a parent to s'. Otherwise, if cumulative cost completely dominates s'', it can be inferred that one can reach s' from s in a better way than s''. Thus, s'' is removed from parents of s' and s is added with the cumulative cost. The fourth possibility occurs when costs of s'' and cumulative costs do not completely dominate each other. In this situation, each cost in cumulative costs is compared with each cost of s'' costs. Equality or domination probabilities causes to remove corresponding cost from its list. At the end of the comparison, s'' is removed from parents of s' if all of its costs are dominated (lines {29-30}) and s is added as a parent if cumulative costs still have non-dominated cost (lines {31-32}).

After organizing parents of s', it is decided to expand it in following iterations. If s is successfully added as a parent and expanding states queue does not already have it, s' is added to the tail of the queue. This can be seen in lines {33-34}.

When all non-dominated parents are properly set from start to goal state, these parents can be followed recursively starting from goal towards start state and multi-objective paths are constructed. Finally, all found paths have non-dominated path costs regarding to each other.

3.3 A Multi Objective Genetic Path Planner : MOGPP

Many real-life optimization problems are NP-hard where optimal solutions could not be found in polynomial time. As evolutionary computing methods are classified as stochastic soft-computing methods and can be applied to NP-hard problems, several genetic algorithms are developed with respect to this problem [17], [18]. To show that MOD* Lite gives feasible and qualified solutions, it is a must to compare it with a stochastic evolutionary method. Thus, a multi objective genetic path planner, MOGPP is also developed in the scope of this study.

3.3.1 Overview

Multi objective genetic path planner (MOGPP) proposed in this study is an alternative solution for finding paths on virtual environments considering multiple objectives. It proposes a classical genetic algorithm structure with crossover and mutation operations where each individual represents a valid path from initial location to target. Experimental and performance results show that MOGPP finds paths in exponential times with respect to MOD* Lite, which are detailed in next section. The chromosome structure, fitness function, crossover, mutation and details of the MOGPP are given in following subsections.

3.3.2 Chromosome Representation

Like all evolutionary algorithms, MOGPP come up with a population where each individual of the population is represented by a chromosome structure. This structure simply stands for a valid path from initial location to target, a legal solution for the problem. Thus, each gene in chromosome is a *cell* in this valid path. As the solution path' s lengths differ from each other, the chromosome lengths may diversify.

3.3.3 Fitness Function

The fitness function is crucial to evaluate a chromosome which is tested for suitability for the environment under some consideration. As the genetic algorithm proceeds, it is expected that the fitness value of the "best" chromosome increases as well as the total fitness of the population as a whole.

The fitness function used in MOGPP is given as follows;

$$F(i) = [\frac{1}{pathLength(i)^2}], [\frac{1}{exposuredRisk(i)^2}]$$

which is the fitness function of individual i in population. This function is represented by a vector of objectives, where first objective is inverse of path length square of corresponding individual and second objective is inverse of calculated exposured risk square during this path. The purpose of fitness function is to yield better results when an individual's path is shorter and safer. On the evaluation process, these fitness values of all individuals in the population are added multi objectively and evaluation value is calculated. The evolution of population is determined by this total fitness value.

3.3.4 Crossover Operation

In genetic algorithms, crossover operation is used to vary the chromosomes and generate new individuals from existing ones. But, to breed new individuals and to use the operators borrowed from natural genetics, the parents should be selected first. There exists many selection operations for genetic algorithms. For MOGPP, roulette-wheel selection method is used to keep the algorithm simple. With this way, greater fitness evaluation owner chromosomes have bigger opportunity to be selected. This mechanism facilitates the population to evolve.

As a single chromosome stands for a valid path in MOGPP, new generated children should also obey this rule. Thus, genetic operations must guarantee that new generated children are consistent and have valid paths. For crossover operation used in MOGPP, assume that two parents' paths are represented by

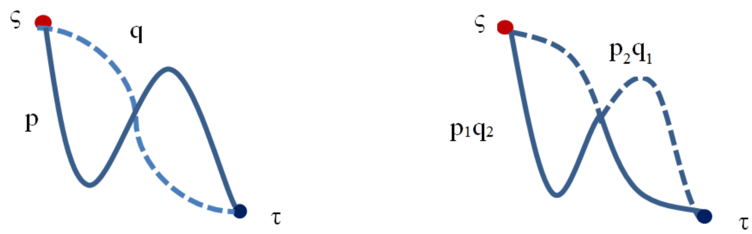
$$P(i) = \{\varsigma, ..., c_{i-1}, c_i, c_{i+1}, ..., \tau\}$$
$$P(j) = \{\varsigma, ..., c_{j-1}, c_j, c_{j+1}, ..., \tau\}$$

for i and j individuals. ς and τ shows initial and target locations, respectively. While crossover, an intersected cell of these paths is determined. Then, each path is split up into two sub-paths referencing by this cell. If this cell is $c_i = c_j$, the splitting operation is done as follows

$$P(i)_1 = \{\varsigma, ..., c_{i-1}\}$$
$$P(i)_2 = \{c_i, c_{i+1}, ..., \tau\}$$
$$P(j)_1 = \{\varsigma, ..., c_{j-1}\}$$
$$P(j)_2 = \{c_j, c_{j+1}, ..., \tau\}$$

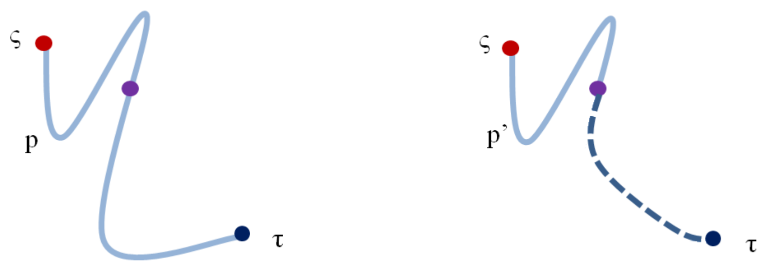
The concatenation of swapped sub-paths generate new individuals

$$P'(i) = \{P(i)_1, P(j)_2\}$$
$$P'(j) = \{P(j)_1, P(i)_2\}$$



(a) Before Crossover (b) After Crossover

Figure 3.4: Crossover Operation



(a) Before Mutation (b) After Mutation

Figure 3.5: Mutation Operation

$P'(i)$ and $P'(j)$ are the new crossovered paths for parents i and j. Also, visualized representation of crossover operation is given in Figure 3.4.

3.3.5 Mutation Operation

In genetic algorithms, mutation is used to maintain genetic diversity of the population. In MOGPP, with respect to given chromosome structure; a cell from corresponding individual' s path is selected randomly first. This cell is the reference point to split up the path into two sub-paths. Then, the sub-path which contains target location is thrown away and a random path to the target is generated instead. The visualization of mutation is given in Figure 3.5.

3.3.6 Details of Algorithm

As mentioned above, MOGPP is designed based on a simple classic genetic algorithm structure. Main loop of MOGPP is given in Algorithm 6.

Algorithm 6 MOGPP : Main Loop

```
function EVOLVE()(P)
    P' = ∅
    P'.addAll(elites(P))
    while P'.size < P.size do
        parents = selectParents(P)
        crossover(parents)
        mutate(parents)
        P'.addAll(parents)
    return P'

function INITIALIZEPOPULATION()
    P = ∅
    for i = 1 → POPULATION_SIZE do
        P(i) = generateRandomPath(ς, τ)
    evaluate(P)
    return P

function PLAN()
    P = initializePopulation()
    while reached to MAX_ITERATION do
        P = evolve(P)
        evaluate(P)
    return bestIndividuals(P)
```

Initialization of algorithm starts with *plan*() function. At first, random valid paths from initial location (ς) to the target (τ) are generated. Notice that these paths do not contain any ties, to simplify and speed up genetic operators' processes.

Each generated path represents an individual in population. These individuals are kept in a directed acyclic graph to cope with multi objectivity. The vertices and edges represent individuals and domination of multi objective path costs of these individuals, respectively. If a path cost of an individual dominates to other's, an edge is established between these individuals' vertices. Non domination and equality do not come up with an edge. When an individual is generated and desired to be added to population, the cost function of this individual is compared with existing individuals' costs and required edge connections are established. The directed acyclic graph structure has the same essence and representation with MOD* Lite' s priority structure, which was detailed in previous section.

After population initialization, all individuals are evaluated with respect to their fitness functions. The evaluation gives better results when an individual's path is shorter and safer. Total fitness value is calculated by adding all individuals' fitness values multi objectively. This value increases while population is evolving.

The evolution process is applied to all individuals of a population. When a population is evolving, predefined number of best individuals are transferred to new population first. Then, two individuals are selected as parents by roulette wheel selection method. This method gives higher chances to the individuals which have better fitness functions to be selected. After two parents are selected, crossover and mutation is applied with predefined distinct probabilities.

After a predefined number of iterations (the maximum iteration count), algorithm is halted and elite individuals are taken as multi objectively best results.

Notice that the amount of initially generated individuals - population size -, maximum iteration count of evolution, number of elite individuals selected on each evolution phase, crossover and mutation probabilities should be predefined and set before the execution of MOGPP.

CHAPTER 4

EXPERIMENTAL RESULTS

MOD* Lite is a domain independent algorithm and can be applied to any virtual environment with given n objectives. These objectives could be whether *maximized* or *minimized*. The vital assumption about objectives is their independence from each other. If two objectives could effect each other in a positive or negative manner, this might reduce or expand objectives vector size, which is out of this work's scope. Thus, we assume that each defined objective is considered in different perspective and can not be transferred to each other.

The algorithm is tested on various environments with different scenarios. However, as it is not possible to exemplify all possible cases, specific and several extreme conditions are selected for experiments. MOD* Lite is compared with MOA* that guarantees optimal solutions in fully observable multi objective environments and MOGPP, a classic genetic solution which can be used for finding paths with multi objective cases.

For testing, all algorithms are implemented in Java language and run under Linux environment which has Intel Core2 Quad CPU running at 2.33GHz and 4 GB of RAM.

All tests are done on 2-D grid maps as detailed in Subsection 3.2.2. In these tests, the agent tries to find available non-dominated best paths with respect to two objectives, path length and risk taken from threat zones. Thus, the agent endeavours to minimize both objectives and tries to find *shortest* and *safest* paths.

For all test cases, several parameters of MOGPP algorithm must be tuned. For instance; number of elitist individuals are 5, population count and maximum iteration are 50, and cross-over and mutation ratios are taken as 0.8 and 0.05, respectively. As MOGPP constructs initial paths randomly, each execution of the algorithm might not give exactly same results at the same execution time even the maps are equal. Thus, all given execution times and selected paths are considered as the average of 10 different executions for MOGPP.

4.1 Fully Observable Tests

First of all, it must be shown that MOD* Lite is complete and gives optimal and/or sub-optimal results in fully observable environments. The performance comparison is done in two dimensions, execution times and paths they generate (path quality), respectively.

In the first set of tests, randomly generated fully observable maps with different sizes (25 x 25, 50 x 50, 75 x 75, 100 x 100, 125 x 125 and 150 x 150), are used. Each of these maps have nearly 30%-32% percent threat zone and 14%-16% percent obstacle ratio. Agent's initial and target locations are also taken randomly. For this case, execution times and generated paths' costs for different sized maps are given in Figure 4.1 and Table 4.1. As seen from path qualities, MOA* finds optimal results and MOD* Lite finds optimal and/or sub-optimal results while gradually increases on the computation time manner. Although MOGPP works on similar times with MOA* especially for large scaled maps, it fails to find optimal or sub-optimal paths. One could set maximum iteration and population count to larger numbers to converge path quality to optimality, but this case increases execution time exponentially. Thus, more modest parameters are chosen for MOGPP to enforce the algorithm to yield reasonable results within expected time.

Notice that taken risk values depend on environmental properties and should not be compared between different sized maps.

In the second set of tests, each algorithm is executed on handcrafted maps with same sizes, threat zone and obstacle ratio with randomized tests as indicated in previous test case. These maps are also assumed fully observable and agent's initial and target locations are taken randomly. All handcrafted test environments *guarantee* that at

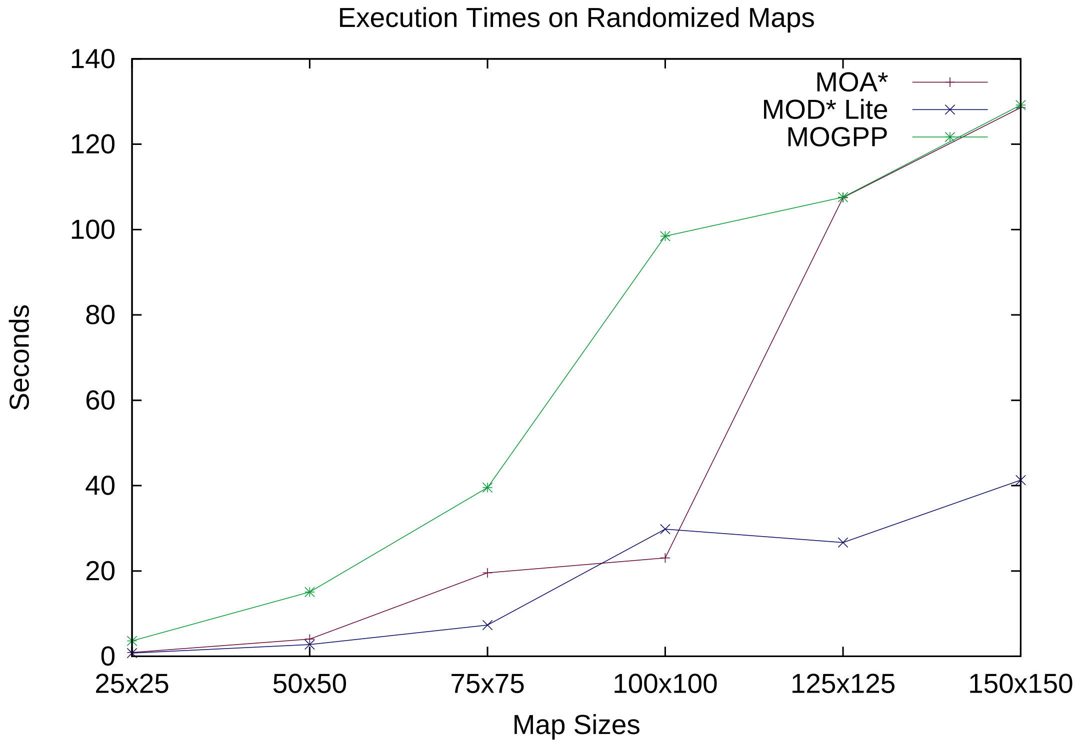


Figure 4.1: Execution Times of Randomly Generated Fully Observable Maps

least two non-dominated paths will be available. Execution times are shown in Figure 4.2 and generated paths' costs are given in Table 4.2.

In execution times in Figure 4.2, it can be seen that MOGPP increases gradually instead of an exponential growth especially on large scaled maps. However, its path quality is not good enough when compared to MOD* Lite and MOA*' s results.

There exists a remarkable point that MOA* has nearly similar results with MOD* Lite in both path quality and execution time. This case shows that even MOD* Lite is based on partially observable dynamic environments, it could also give *as good results as* MOA* on stationary and fully observable environments and can be applied on those environments.

Table 4.1: Non-dominated Path Costs For Randomized Maps

Map Size	MOD* Lite	MOA*	MOGPP
25 x 25	(49, 571)	(49, 571)	(49, 3523)
	(51, 10)	(51, 10)	(51, 764)
			(53, 58)
50 x 50	(99, 982)	(99, 982)	(103, 7399)
	(101, 0)	(101, 0)	(121, 178)
			(123, 0)
75 x 75	(149, 1549)	(149, 115)	(175, 730)
	(151, 221)	(153, 0)	(187, 272)
	(153, 0)		(201, 0)
100 x 100	(199, 10)	(199, 10)	(301, 2357)
	(201, 4)	(201, 4)	(329, 1087)
	(203, 0)	(203, 0)	(349, 746)
125 x 125	(90, 1036)	(90, 1036)	(225, 10677)
	(92, 293)	(92, 293)	(239, 2249)
	(94, 165)	(94, 165)	
	(96, 101)	(96, 101)	
150 x 150	(126, 128)	(126, 128)	(191, 14720)
			(207, 5316)
			(211, 312)

4.2 Partially Observable Tests

As discussed in previous sections; the main difference of MOD* Lite from existing classic path planning or evolutionary based algorithms is its adaptivity to partially observable dynamic environments. To show this advantage, partially observable tests are done with randomized maps of sizes 75 x 75, 100 x 100 and 125 x 125. On these maps, agent's initial and target locations are chosen to be the furthermost cells in the environment. For each map, agent' s sensor range was set between 20% to 60% and execution times were observed. An example search space of MOD* Lite with 30% sensor range can be seen in Figure 4.3. In this figure, agent is depicted with a cyan dot. The fogged gray area represents agent' s sensor range and drawn purple path through temporary goal (blue dot) can be seen.

In these tests, the agent starts to plan a path towards the nearest available cell within its sensor range -the temporary goal- to the actual goal with respect to Manhattan Distance. After planning, consider that agent has found three paths with costs $(15, 200)$, $(18, 230)$ and $(23, 260)$. In such cases, the agent tends to choose the path with cost

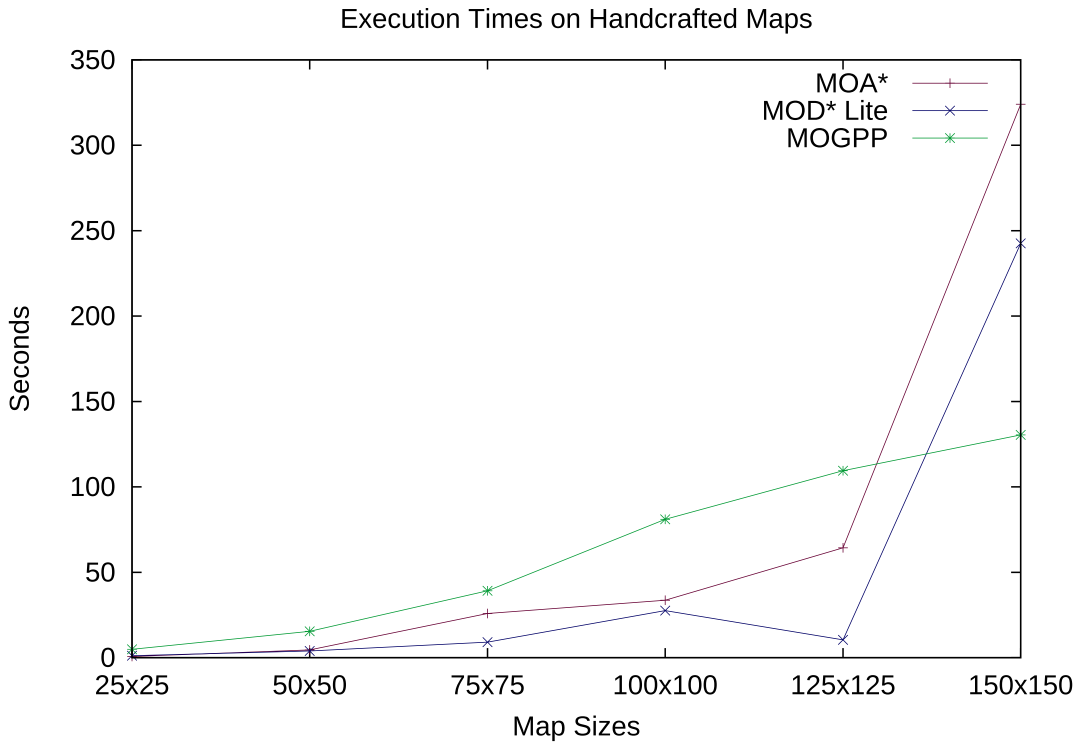


Figure 4.2: Execution Times of Handcrafted Fully Observable Maps

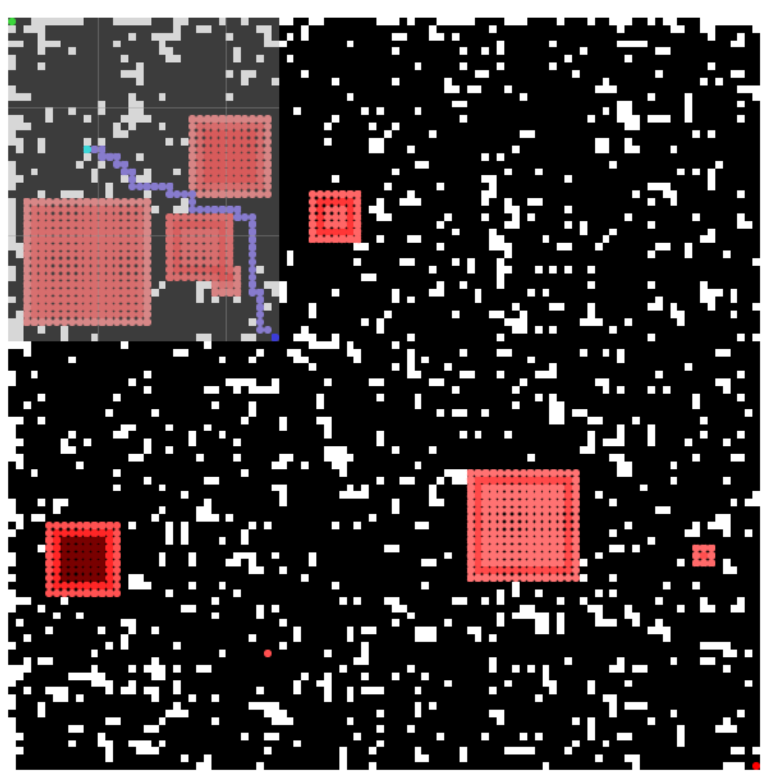


Figure 4.3: A 100x100 Partially Observable Map with 30% Sensor Range

Table 4.2: Non-dominated Path Costs For Handcrafted Maps

Map Size	MOD* Lite	MOA*	MOGPP
25 x 25	(49, 722)	(49, 722)	(53, 1227)
	(51, 505)	(51, 505)	(56, 938)
	(55, 216)	(55, 216)	(59, 216)
50 x 50	(99, 14)	(99, 14)	(101, 1871)
	(101, 0)	(101, 0)	(125, 1156)
			(151, 736)
75 x 75	(149, 1927)	(149, 1927)	(235, 737)
	(151, 1329)	(151, 1329)	(261, 372)
	(153, 279)	(153, 279)	(279, 0)
	(159, 0)	(159, 0)	
100 x 100	(199, 1077)	(199, 144)	(297, 4390)
	(201, 20)	(201, 20)	(315, 3710)
	(205, 0)	(205, 0)	(395, 1861)
125 x 125	(249, 15)	(249, 15)	(391, 9516)
		(257, 0)	(445, 5358)
			(451, 365)
			(561, 0)
150 x 150	(301, 20)	(299, 145)	(266, 6110)
		(301, 20)	(270, 5915)
		(315, 0)	(284, 4530)

$(18, 230)$, the median of paths. This ad-hoc strategy could be set explicitly according to the domain that the algorithm works on. Afterwards, it starts to follow the chosen path. When new cells are available or a weight of a cell is changed within sensor range, agent reassigns the temporary goal and re-executes the path planner algorithm. This process iterates until the agent reaches to the desired goal location.

The fundamental advantage of MOD* Lite can be seen very clearly on these tests. While MOD* Lite has the capability of updating only the effected states due to its incremental nature, MOA* re-plans the overall path from scratch when new parts become known and the weights of some cells have changed. This situation causes MOA* and MOGPP to work on exponentially long times. Total execution times to reach to the target for these test cases are given in Figures 4.4, 4.5 and 4.6. As can be seen from results, MOD* Lite can easily handle the dynamical issues of the environment where MOA* and MOGPP fails. Due to discovering different parts of the environment during execution, actual traversed path's costs of MOD* Lite, MOA* and MOGPP might be slightly different from each other, where MOD* Lite could follow

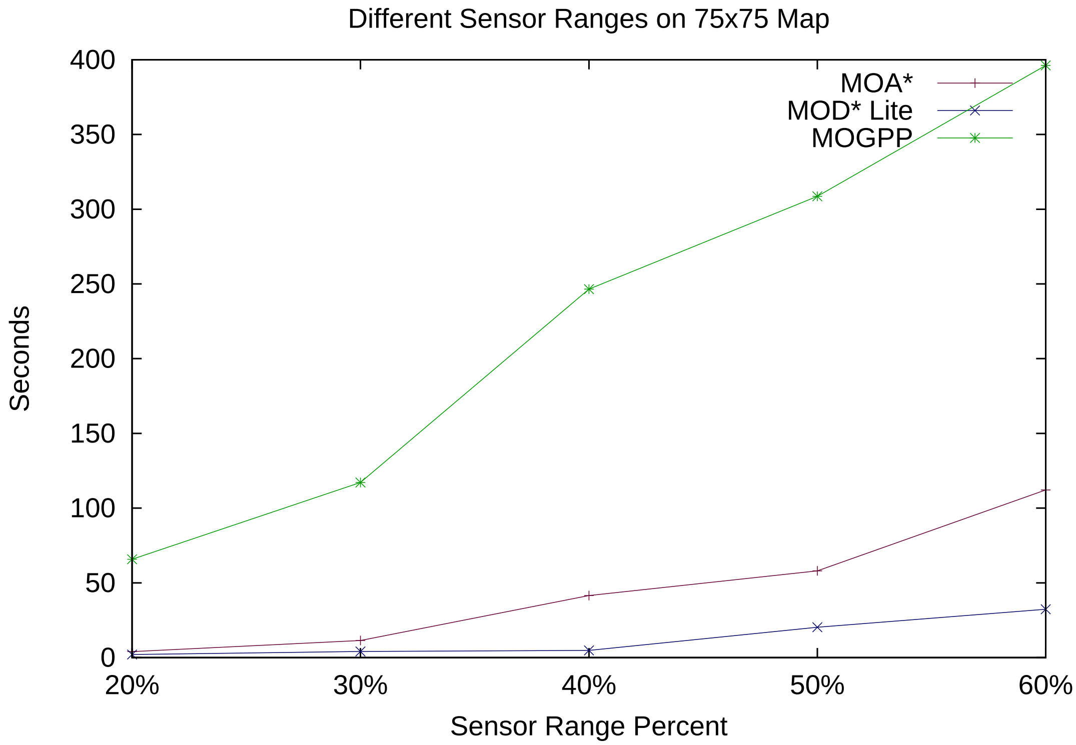


Figure 4.4: 75x75 Partially Observable Map on Different Sensor Ranges

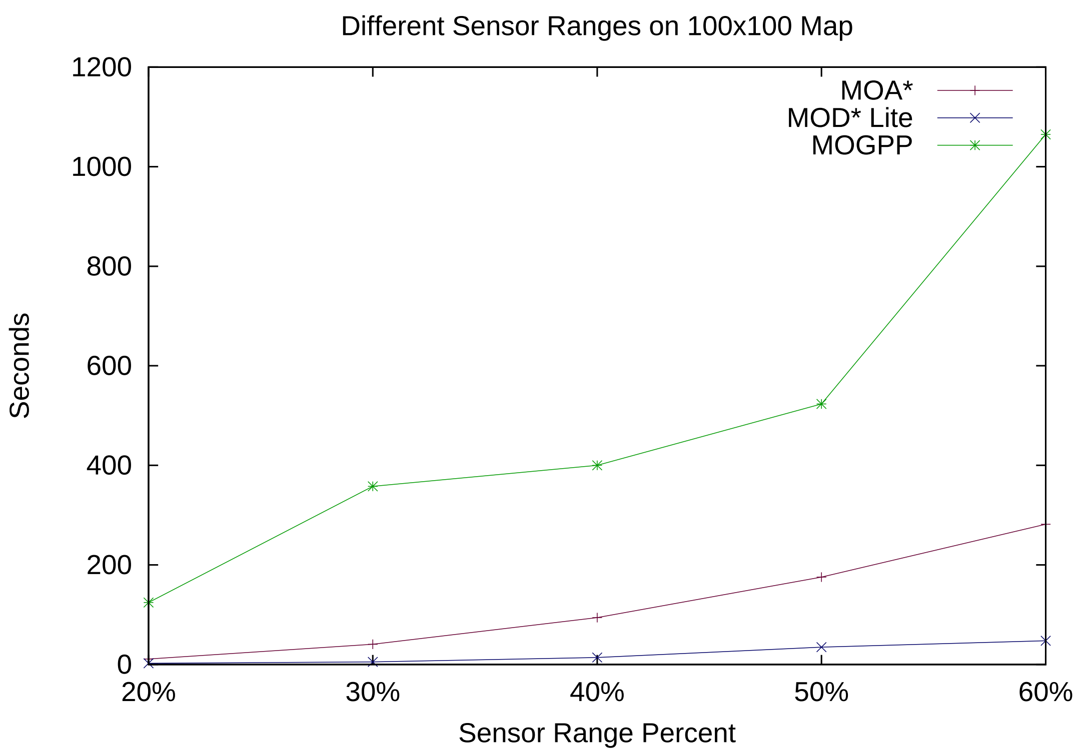


Figure 4.5: 100x100 Partially Observable Map on Different Sensor Ranges

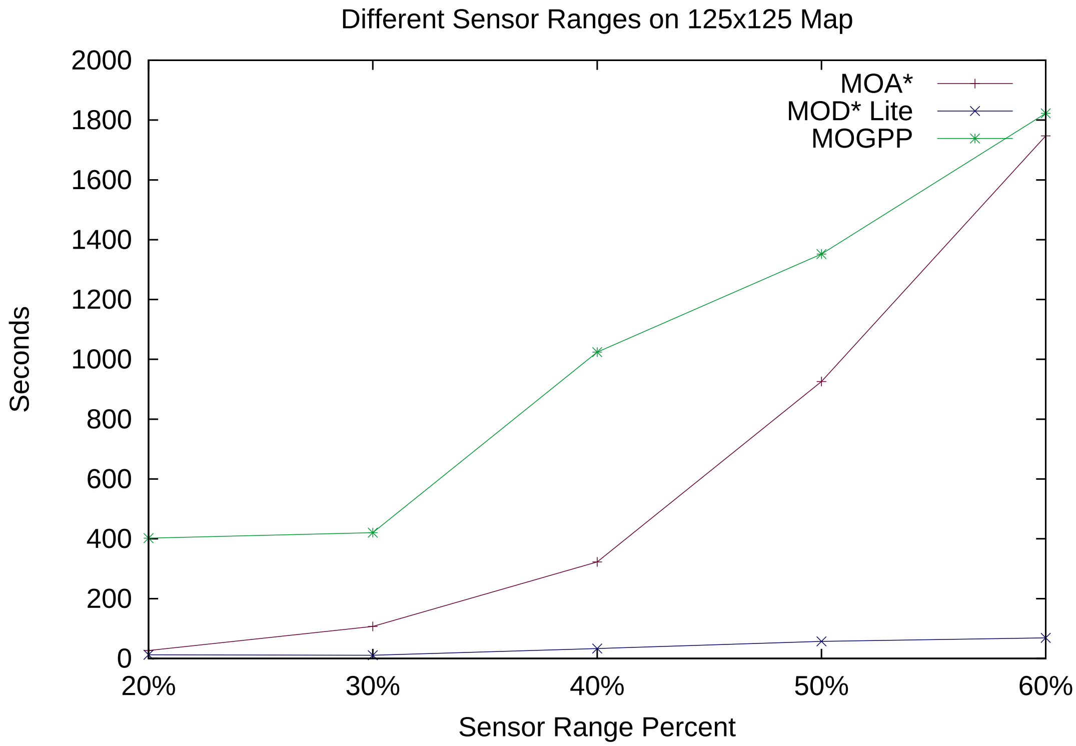


Figure 4.6: 125x125 Partially Observable Map on Different Sensor Ranges

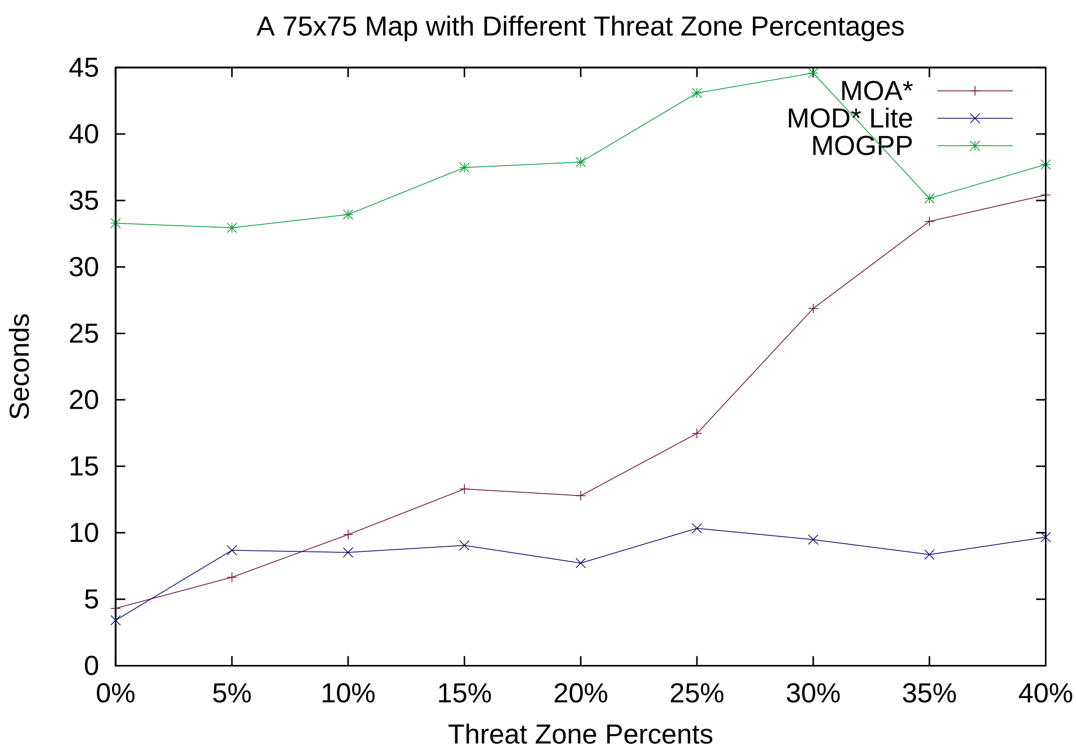


Figure 4.7: Execution times of 75x75 Fully Observable Map on Different Threat Zone Percents

a better path with respect to MOA* or MOGPP or vice versa.

4.3 Multi Objectivity Tests

As threat zones and their risk values are used as the second objective, percentages of these zones also effect execution time and generated path quality. In this set of tests different threat zone percents are tested on a fully observable 75 x 75 map and results are given in Figure 4.7. It could be observed that increasing one of the objective' s ratio, or risks of threat zones for this test, does not effect performance of MOD* Lite and MOGPP too much, they find results in approximately similar times. However, MOA* is tightly coupled with it and execution time increases gradually as the threat percentage increases.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Searching, path planning and navigation in virtual environments are vital issues and referred by many visualized real-world applications. These are recent problems and are considered in areas including robotics, virtual simulations or computer games and has been studied for many years. Computer science came up with many solutions considering on single criteria; i.e. shortest path from an initial location to a target location or safest path between two transition points. However, more complex situations where considering multiple objectives instead of a single objective for searching and finding an optimal or sub-optimal solution become more appropriate when it is desired to apply searching and navigation techniques to real world applications. On the other hand, one should not ignore interactive behaviour and dynamics of the real world and consider them for more realistic modelling. In a nutshell, a multi objective search algorithm which can work on partially observable dynamic environments is required to satisfy all of these requirements.

With this motivation; a novel approach for searching, planning and finding paths on known and unknown partially observable dynamic environments where the agent needs to optimize more than one criteria that cannot be transformed to each other, MOD* Lite, is presented in this thesis study. MOD* Lite is based on D* Lite and it brings multi objectivity to the solution space successfully, which is required in many real-world problems. It is a domain independent algorithm and could be applied to any partially/fully observable dynamic virtual environment with n different non-interacting objectives. It is compared with known and complete multi objective off-line path planning algorithm, MOA*, and also with a novel evolutionary solution, multi objective genetic path planner, MOGPP, based on solution quality and execution times. Experimental results show that MOD* Lite is able to optimize path quality and is fast enough to be used in real-world multi objective application areas such as robotics, computer games, and virtual simulations. According to the conducted literature survey and knowledge gained, MOD* Lite is the only and the one incremental search and navigation algorithm which could handle multi-objectivity.

There is an obvious gap on moving target multi objective path planning area. As there exists several incremental moving target solutions for virtual environment proposed within recent years [22], [23], [24]; a modified versions of these solutions can be used with multi objectivity. Also MOD* Lite could be reconsidered in multi agent perspective where each agent distributively execute their planners and cooperate to reach a target location. Thus, further studies will include incremental moving target multi objective problems, their solutions and multi-agent perspective.

REFERENCES

[1] Serhat Bayili and Faruk Polat. Limited-damage a*: A path search algorithm that considers damage as a feasibility criterion. *Knowledge-Based Systems*, 24(4):501 – 512, 2011.

[2] Nabeel Bukhari, Ayesha Khan, Abdul Rauf Baig, and Kashif Zafar. Optimization of route planning using simulated ant agent system. *International Journal of Computer Applications*, 4(8):1–4, August 2010. Published by Foundation of Computer Science.

[3] Oscar Castillo, Leonardo Trujillo, and Patricia Melin. Multiple Objective Genetic Algorithms for Path-planning Optimization in Autonomous Mobile Robots. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 11(3):269–279–279, February 2007.

[4] Carlos A. Coello. An updated survey of ga-based multiobjective optimization techniques. *ACM Comput. Surv.*, 32:109–143, June 2000.

[5] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. J.Wiley & Sons, 1st edition, June 2001.

[6] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] G. Dozier, S. McCullough, A. Homaifar, E. Tunstel, and L. Moore. Multiobjective evolutionary path planning via fuzzy tournament selection. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 684 –689, May 1998.

[8] Jung Leng Foo, Jared Knutzon, Vijay Kalivarapu, James Oliver, and Eliot Winer. Three-dimensional path planning of unmanned aerial vehicle in a virtual battlespace using b-splines and particle swarm optimization. *Journal of Aerospace Computing Information and Communication*, 6(April):271–290, 2009.

[9] Feng Guo, Hongrui Wang, and Yantao Tian. Multi-objective path planning for unrestricted mobile. In *Automation and Logistics, 2009. ICAL '09. IEEE International Conference*, pages 1046 –1051, 2009.

[10] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, July 1968.

[11] Sven Koenig and Maxim Likhachev. D*lite. In *Eighteenth national conference on Artificial intelligence*, pages 476–483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[12] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong planning a*. *Artif. Intell.*, 155(1-2):93–146, May 2004.

[13] Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2–3):189 – 211, 1990.

[14] I.M. Mitchell and S. Sastry. Continuous path planning with multiple constraints. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 5, pages 5502 – 5507 Vol.5, 2003.

[15] Amin Zargar Nasrollahy and Hamid Haj Seyyed Javadi. Using particle swarm optimization for robot path planning in dynamic environments with moving obstacles and target. *Computer Modeling and Simulation, UKSIM European Symposium on*, 0:60–65, 2009.

[16] Tuğcem Oral and Faruk Polat. A multi-objective incremental path planning algorithm for mobile agents. In *Proceedings of the 2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, WI-IAT '12. IEEE Computer Society, 2012. Will be published by IEEE Computer Society in December 2012.

[17] J.M.A. Pangilinan and G.K. Janssens. Evolutionary algorithms for the multiobjective shortest path problem. volume 21 (PWASET), pages 205–210.

[18] X Peng, D Xu, and F Zhang. *UAV online path planning based on dynamic multiobjective evolutionary algorithm*, pages 5424–5429. 2011.

[19] Anthony Stentz. The focussed d* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, pages 1652–1659, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[20] Anthony (Tony) Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, pages 3310 – 3317, May 1994.

[21] Bradley S. Stewart and Chelsea C. White, III. Multiobjective a*. *J. ACM*, 38(4):775–814, October 1991.

[22] Xiaoxun Sun, William Yeoh, and Sven Koenig. Efficient incremental search for moving target search. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 615–620, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[23] Xiaoxun Sun, William Yeoh, and Sven Koenig. Generalized fringe-retrieving a*: faster moving target search on state lattices. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 1081–1088, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[24] Xiaoxun Sun, William Yeoh, and Sven Koenig. Moving target d* lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 67–74, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[25] Zbigniew Tarapata. Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *Int. J. Appl. Math. Comput. Sci.*, 17:269–287, June 2007.

[26] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.