MOD* Lite: An Incremental Path Planning Algorithm Taking Care of Multiple Objectives

Tugcem Oral and Faruk Polat

***Abstract*—The need for determining a path from an initial location to a target one is a crucial task in many applications, such as virtual simulations, robotics and computer games. Almost all of the existing algorithms are designed to find optimal or sub-optimal solutions considering only a single objective, namely path length. However, in many real life application path length is not the sole criteria for optimization, there are more than one criteria to be optimized that cannot be transformed to each other. In this paper, we introduce a novel multi objective incremental algorithm, MOD* Lite built upon a well-known path planning algorithm, D* Lite. A number of experiments are designed to compare the solution quality and execution time requirements of MOD* Lite with the multi objective A* algorithm (MOA*) and an alternative genetic algorithm solution.**

I. Introduction

The problem of finding a path for an autonomous agent from an initial location to a destination location is a popular task in real-world applications including robotics, virtual simulations or computer games and has been studied for many years. Existing path planning algorithms can be classified into four classes: off-line algorithms [1] [2], on-line algorithms [3], incremental algorithms [4], [5], [6] and soft computing solutions [7], [8]. 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 search algorithms try to merge advantages of both approaches to obtain better execution time without sacrificing optimality much. 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 problems 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 limited fuel resource, or time thresholds.

Department of Computer Engineering, Middle East Technical University, 06800 Ankara, Turkey (tugcem.oral@gmail.com, polat@ceng.metu.edu.tr)

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 decision making under multiple criteria at the same time. In this paper, an incremental path finding algorithm called Multi-Objective D* Lite (MOD* Lite) which extends an existing incremental algorithm, Dynamic A* Lite (D* Lite) [5], is introduced. MOD* Lite [9] 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 observes the environment partially, and furthermore needs to optimize multiple objectives at the same time.

In order to evaluate the performance of MOD* Lite, we also developed a multi objective genetic path planning (MOGPP) algorithm. This algorithm finds initial paths randomly and its population that contains solution alternatives evolves according to a fitness function. MOD* Lite is both compared against MOGPP and MOA* algorithm [10], 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.

This paper is organized as follows: Section 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 section. The problem definition, characteristics of the environment and proposed solutions (MOD* Lite and MOGPP) are presented in Section 3 and 4. Experimental studies and their results are stated in Section 5. Finally, the conclusion and future studies are provided in Section 6.

II. Related Work & Background

In the literature, there are several algorithms focusing on path planning. In this section, existing studies are introduced relevant to our work where the motivation is to handle exis-

tence of multiple objectives and partial observability in path planning.

In [11], Bayili and Polat introduced a multi-objective path planning algorithm, Limited Damage A*, 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 [7]. He gave a classification of multi-objective shortest path (MOSP) problems and discussed different models of them. He also 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 [12]. 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 [13], 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. [8] 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) [14] 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-dominated optimal solutions become obstructions.

Castillo et al. also worked on evolutionary algorithms for MOPP in their study [15]. 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 [16]. 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 optima [17]. 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 [18]. 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 [19]. Also [20], 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* [4] which guarantees to be optimal

and is functionally equivalent to re-planning from scratch. Later, D* Lite was proposed by Koenig et al. [5] which utilized the same navigation strategy with D* but algorithmically different. It was based on Lifelong Planning A* (LPA*) [21]. 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*.

A. Multi Objective A* (MOA*)

Classical A* [2] is a complete and optimal solution for the cases where only a single optimization criterion 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*) [10] 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* (explored nodes) and *CLOSED* (expanded 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.

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, each newly 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.

B. D* Lite

D* Lite [5] is one the of most popular goal-directed navigation algorithms and widely used in unknown environment. It is an adaptation of Lifelong Planning A* [21] which is an incremental derivation of A* [2]. It determines the same paths as D* Algorithm [4] 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. The following subsections inclued a detailed review of D* Lite as our proposed solution is built upon this successful incremental path planning algorithm.

1) Overview: Consider an agent's path planning and navigation task in a dynamic unknown environment. In this scenario, the agent always observes its current cell's neighbor cells and try to move one of them if it is traversable. The agent starts from an initial cell and moves through to the goal cell. It always tries 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 follows the 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 **removed** contains a sample environment depicting 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 re-planning path just according to these changed cells.

2) 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 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.

3) 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

Algorithm 1 D* Lite Outline

```
1: function CALCULATEKEY(s)
2:     return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];

3: function INITIALIZE()
4:     U = ∅;
5:     k_m = 0;
6:     for all s ∈ S do
7:         rhs(s) = g(s) = ∞;
8:     rhs(s_goal) = 0;
9:     U.insert(s_goal, calculateKey(s_goal));

10: function UPDATEVERTEX(u)
11:     if u ≠ s_goal then
12:         rhs(u) = min_{s'∈succ(u)}(c(u, s') + g(s'));
13:     if u ∈ U then U.remove(u);
14:     if g(u) ≠ rhs(u) then
15:         U.insert(u, calculateKey(u));

16: function COMPUTESHORTESTPATH()
17:     while U.topKey() < calculateKey(s_start) || rhs(s_start) ≠ g(s_start) do
18:         k_old = U.topKey();
19:         u = U.pop();
20:         if k_old < calculateKey(u) then
21:             U.insert(u, calculateKey(u));
22:         else if g(u) > rhs(u) then
23:             g(u) = rhs(u);
24:             for all s ∈ pred(u) do updateVertex(s)
25:         else
26:             g(u) = ∞;
27:             for all s ∈ pred(u) ∪ u do updateVertex(s);
```

Algorithm 2 Cont'd of D* Lite Outline

```
1: function PLAN()
2:     s_last = s_start
3:     initialize();
4:     computeShortestPath();
5:     while s_start ≠ s_goal do
6:         if g(s_start) = ∞ then there is no known path
7:         s_start = argmin_{s'∈succ(s_start)}(c(s_start, s') + g(s'));
8:         Move to s_start;
9:         Scan the graph for changed edge costs;
10:        if Any weight cost changed then
11:            k_m = k_m + h(s_last, s_start));
12:            s_last = s_start;
13:            for all directed edges (u,v) with changed edge costs do
14:                Update the edge cost c(u,v);
15:                updateVertex(u);
16:            computeShortestPath();
```

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 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.

III. Multi Objective D* Lite : MOD* Lite

A. Motivation

Assume that an unmanned aerial vehicle (UAV) is taking off from an initial location. Its goal is to discover an enemy unit on a predefined target location in an unknown dynamic environment. However, the enemy unit is protected by air defense units scattered on the terrain having different capabilities (hit ratios) and coverage areas. Each defense 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 defense 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 monitor the target quickly but the risk of being hit by an air defense unit must be minimized. This means that the UAV should find both the 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 needs to re-plan the current path when an unknown part of the environment become known, or known parts are changed (i.e., a visible defense unit's coverage area changes, shrinks or enlarges; or defense unit is disabled / enabled) as it navigates. Mostly, evolutionary search algorithms focus on this issue and come up with several solutions like [22], [23]. 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.

In this study, a multi objective incremental path planning algorithm is developed based on non-optimized version of D* lite [5] to meet these requirements. The algorithm is called multi objective D* lite, or MOD* Lite [9]. To prove effectiveness of MOD* Lite, an alternative solution based that makes use of evolutionary computation, Multi Objective Genetic Path Planner (MOGPP) is also developed.

B. Overview

Multi-objective problems focus on considering more than one objective concurrently (at the same time). Consequently, *all* objectives that are expressed with scalar values and atomic operations (like addition, checking for equality, etc.) on them need to be adapted to vectors of scalars and operations on them. 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 is dominated by v), $u > v$ (u dominates v) and $u = v$ there is a fourth alternative 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;

$$u_i = v_i \ \ \forall i \ 1 \leq i \leq n$$

where n is the number of objectives for $u = [u_1, \cdots, u_n]$ and $v = [v_1, \cdots, v_n]$.

We say that u *dominates* v if u is better than v in at least one objective. In other words, there is no objective of v which 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 $u = [3, 4], v = [4, 6], w = [6, 2]$. Here u dominates v, but u and w 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 other objectives. For the sake of simplicity, we assume that there are two objectives to be minimized and describe MOD* Lite accordingly. Note that the algorithms can easily be instantiated to work with more than two objectives. For the experimental study, MOD* Lite is is realized in 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.

C. 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 $(2v+1)\times(2v+1)$, 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.

D. The Components and Variables

MOD* Lite is the multi-objective extension of D* Lite. It can be applied to any 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 base vector with n values corresponding to n objectives. Any value could be 0 or ∞ whether the relevant objective is to be minimized or maximized, 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 $u = [u_1, \cdots, u_n]$ *completely dominates* $v = [v_i, \cdots, v_n]$ iff $\forall i \ u_i \leq v_i$ and $\exists j \quad u_j < v_j$ where $1 \leq i, j \leq n$.

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.

Note that "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 to as locally underconsistent if $g(s) < rhs(s)$ or locally overconsistent if $g(s) > rhs(s)$. In 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

Algorithm 3 Main loop of MOD* Lite

```
1: function CALCULATEKEY(s)
2:     k2(s)= nonDom(g(s), rhs(s))
3:     k1(s) = sum(h(s_start, s), k_m, k2(s))
4:     return [k1(s), k2(s)]

5: function INITIALIZE()
6:     U = ∅
7:     k_m = ObjectiveVector.MIN
8:     for all s ∈ S do
9:         rhs(s)=g(s)=ObjectiveVector.MAX
10:    rhs(s_goal) = ObjectiveVector.MIN
11:    U.insert(s_goal, calculateKey(s_goal))

12: function PLAN()
13:    initialize()
14:    computeMOPaths()
15:    while true do
16:        solutionPaths = generateMOPaths()
17:        if solutionPaths = null then there is no known path
18:        Wait for any weight cost to change;
19:        if Any weight cost changes then
20:            k_m = sum(k_m, h(s_goal, s_start))
21:            for all Changed weight costs of edges(u,v) do
22:                Update cost c(u,v)
23:                updateVertex(u)
24:            computeMOPaths()
```

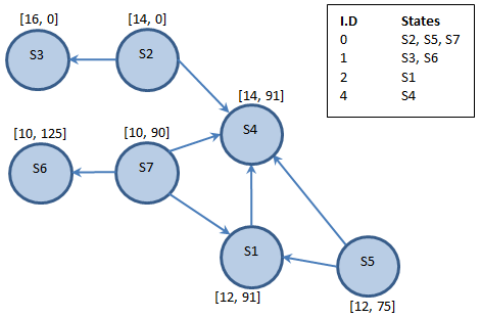


Fig. 1. Directed Acyclic State Expansion Graph

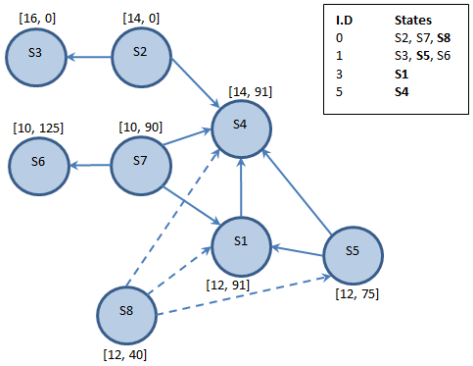


Fig. 2. State Expansion Graph after Adding State $S8$

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 a 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 is introduced in case of 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 nodes with incoming degree 0 are the non-dominated nodes 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 corresponds to the number of states dominated by s. Also, these two methods could be combined where two topologically ordered lists, one for minimum incoming degrees and the other 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. The strategy to be applied can be selected according to application domain requirements. We preferred selecting and removing from minimum incoming degrees list for our application domain. 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 1. 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 2. $S8$ 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 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.

E. 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

```
1:  function UPDATEVERTEX(u)
2:      if u ≠ s_goal then
3:          rhs(u) = nonDom_{s'∈succ(u)}(sum(c(u, s'), g(s')))
4:      if u ∈ U then U.remove(u)
5:      if !equals(g(u), rhs(u)) then
6:          U.insert(u, calculateKey(u))

7:  function COMPUTEMOPATHS()
8:      while dominatesAll(calculateKey(s_start), U.topKeys()) do
9:          k_old = U.topKey()
10:         u = U.pop()
11:         k_new = calculateKey(u)
12:         if k_old.completelyDominates(k_new) then
13:             U.insert(u, k_new)
14:         else if rhs(u).completelyDominates(g(u)) then
15:             g(u) = rhs(u)
16:             for all s ∈ pred(u) do updateVertex(s)
17:         else if g(u).completelyDominates(rhs(u)) then
18:             g(u) = ObjectiveVector.MAX
19:             for all s ∈ pred(u) ∪ {u} do updateVertex(s)
20:         else
21:             g(u) = nonDom(g(u), rhs(u))
22:             for all s ∈ pred(u) do updateVertex(s)
```

F. 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 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,

Algorithm 5 Path Generator Algorithm

```
1: function GENERATEMOPATHS()
2:     expandingStates.add(s_start)
3:     while !expandingStates.isEmpty() do
4:         s = expandingStates.poll()
5:         nonDomSuccs = nonDom_{s'∈succ(s)}(sum(c(s, s'), g(s'))
6:         for all s' ∈ nonDomSuccs do
7:             if s.parents() = null then
8:                 s'.parents().put(s, c(s, s'))
9:             else
10:                cumulativeC = sum(c(s, s'), s.parents().values())
11:                if s'.parents() = null then
12:                    s'.parents().put(s, cumulativeC)
13:                else
14:                    for all s'' ∈ s'.parents() do
15:                        if equals(s'.parents(s''), cumulativeC) OR
   completelyDominates( s'.parents(s''), cumulativeC) then
16:                            break
17:                        else if completelyDominates(cumulativeC,
   s'.parents(s'')) then
18:                            s'.parents().remove(s'')
19:                            s'.parents().put(s, cumulativeC)
20:                        else
21:                            for all cC ∈ cumulativeC do
22:                                for all eC ∈ s'.parents(s'') do
23:                                    if eC.equals(cC) OR
   eC.dominates(cC) then
24:                                        cumulativeC.remove(cC)
25:                                        break
26:                                    else if cC.dominates(eC) then
27:                                        s'.parents(s'').remove(eC)
28:                                        break
29:                                if s'.parents(s'') = null then
30:                                    s'.parents().remove(s'')
31:                            if !cumulativeC = null then
32:                                s'.parents().put(s, cumulativeC)
33:            if s'.parents.contains(s) AND !expand-
   ingStates.contains(s') then
34:                expandingStates.add(s')
35:    solutionPaths = construct paths recursively traversing parents
36:    return solutionPaths
```

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.

IV. A Multi Objective Genetic Path Planner : MOGPP

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 an initial location to a target. Experimental and performance results show that MOGPP finds paths in exponential times with respect to MOD* Lite, which are detailed in next section.

A chromosome corresponds to a valid path from an initial location to a target one specifying a legal solution for the problem. Thus, each gene in chromosome is a *cell* in this valid path. As the solution path's differ from each other, the chromosome lengths may vary.

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 the first objective is the inverse of the square of the path length of the corresponding individual and the second objective is the inverse of the square of the exposed risk of 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.

A. Crossover and Mutation Operations

We used roulette-wheel selection mechanism to determine parents for mating (crossover operation). As a single chromosome stands for a valid path in MOGPP, newly generated children should also obey this rule. Thus, genetic operations must guarantee that newly generated children are consistent and exhibit valid paths. For crossover operation used in MOGPP, assume that two parents' paths are represented by

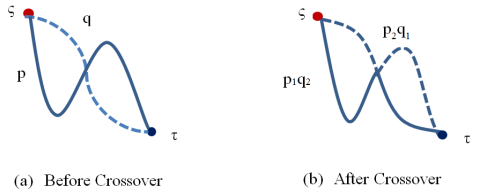
$$P(i) = \{\varsigma, ..., c_{i-1}, c_i, c_{i+1}, ..., \tau\}$$
$$P(j) = \{\varsigma, ..., c_{j-1}, c_j, c_{j+1}, ..., \tau\}$$

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

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

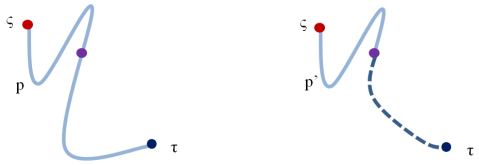
The concatenation of swapped sub-paths generate new individuals

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



(a) Before Crossover (b) After Crossover

Fig. 3. Crossover Operation



(a) Before Mutation (b) After Mutation

Fig. 4. Mutation Operation

$P'(i)$ and $P'(j)$ are the new paths for parents i and j. Also, a visualization of crossover operation is given in Figure 3.

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 4.

B. 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

```
1:  function EVOLVE()(P)
2:      P' = ∅
3:      P'.addAll(elites(P))
4:      while P'.size < P.size do
5:          parents = selectParents(P)
6:          crossover(parents)
7:          mutate(parents)
8:          P'.addAll(parents)
9:      return P'

10: function INITIALIZEPOPULATION()
11:     P = ∅
12:     for i = 1 → POPULATION_SIZE do
13:         P(i) = generateRandomPath(ς, τ)
14:     evaluate(P)
15:     return P

16: function PLAN()
17:     P = initializePopulation()
18:     while reached to MAX_ITERATION do
19:         P = evolve(P)
20:         evaluate(P)
21:     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 terminated 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.

V. Experimental Results

The algorithm is tested on various environments with different specifications. MOD* Lite is compared with MOA* that guarantees optimal solutions in fully observable multi objective problem settings. Our aim is to test the quality of solutions produced by MOD* Lite with respect to optimal ones generated by MOA* ignoring its huge running time costs. Note that MOA* requires exponential time and cannot be used for unknown or partially known environments. Furthermore, we compared MOA* with MOGPP, a classic genetic solution which can be used for finding paths with multi objective cases.

All algorithms are coded in Java and tested on a Linux environment with hardware Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz and 8 GB of RAM with -Xmx6192m parameter for JVM.

All tests are done on 2-D grid maps as detailed in Subsection III-C. 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 endeavors 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; the number of elite individuals is taken as 5, population count and maximum number of iterations 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 the 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 runs of MOGPP.

A. Fully Observable Tests

First of all, it must be shown that MOD* Lite is complete and gives optimal and/or acceptably 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 $m \times m$ (20 x 20, 40 x 40, 60 x 60, 80 x 80, 100 x 100, 120 x 120, 140 x 140 and 160 x 160), are used. Each of these maps have nearly 25%-30% percent threat zone and 14%-16% percent obstacle ratio. Agent's initial and target locations are taken as farmost cells (1×1 and $m \times m$) on the diagonal. For this case, execution times and generated paths' costs for different sized maps are given in Figure 5 and Table I. Note that in Table I, solutions reported by the algorithms are given as pairs of two numbers, the first one is the length of the path, the second one is the accumulated risk value through that path, both of which to be minimized. Concerning solution quality, MOA* finds optimal results and MOD* Lite finds optimal or acceptably sub-optimal results while its running times are better than those of MOA*. Although MOGPP works on similar times with MOD* Lite, it fails to find optimal or sub-optimal paths especially for large scaled maps.We could set maximum iteration and population count to larger numbers to have converge on path quality, but this would increases execution times 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 of the 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 as farmost cells on diagonal. All handcrafted test environments *guarantee* that at least two non-dominated paths will be available. Execution times are shown in Figure 6 and generated paths' costs are given in Table II.

In Figure 6, 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.

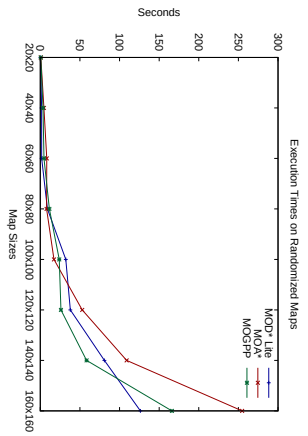


Fig. 5. Execution Times of Randomly Generated Fully Observable Maps

TABLE I
NON-DOMINATED PATH COSTS FOR RANDOMIZED MAPS

Map Size	MOD* Lite	MOA*	MOGPP
20 x 20	(39,90) (43,0)	(39,90) (43,0)	(41,916) (43,408)
40 x 40	(79,9) (81,0)	(79,9) (81,0)	(89,2789) (103,1239) (105,296)
60 x 60	(119,0)	(119,0)	(151,1970) (169,549) (181,426)
80 x 80	(159,198) (161,0)	(159,58) (161,0)	(213,6382) (237,2007) (263,1581) (269,955) (285,942)
100 x 100	(199,885) (201,708) (203,0)	(199,885) (201,708) (203,0)	(285,15804) (295,14130) (299,14099) (305,10979) (341,9851) (377,177)
120 x 120	(239,0)	(238,0)	(371,12817) (399,7346)
140 x 140	(279,45) (281,42)	(279,45) (285,12) (303,0)	(445,10920) (483,5281) (515,4000) (517,3520)
160 x 160	(319,0)	(319,0)	(545,7530) (547,4602)

There exists a remarkable point that MOA* has nearly similar results with MOD* Lite in path quality. This case shows that even MOD* Lite is developed for partially observable dynamic environments, it could also give *as good results as* MOA* on stationary and fully observable environments and can be used for off-line use.

B. Partially Observable Tests

As discussed in previous sections; the main difference of MOD* Lite from existing classic path planning or evolution-

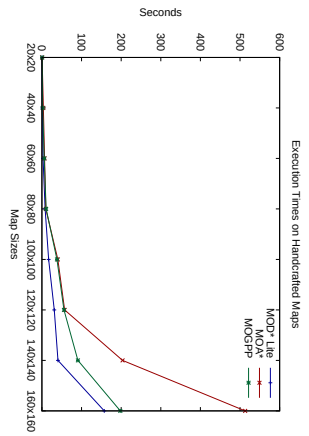


Fig. 6. Execution Times of Handcrafted Fully Observable Maps

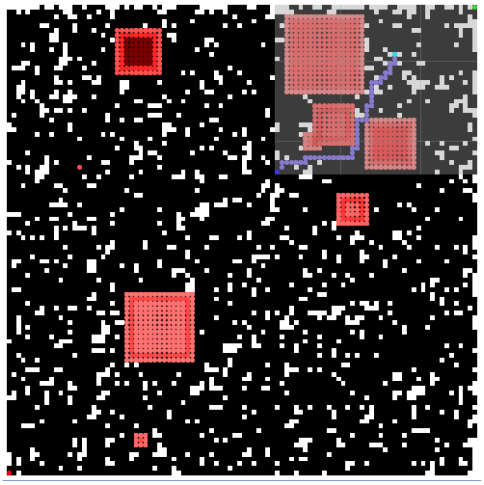


Fig. 7. A 100x100 Partially Observable Map with 30% Sensor Range

TABLE II
NON-DOMINATED PATH COSTS FOR HANDCRAFTED MAPS

Map Size	MOD* Lite	MOA*	MOGPP
20 x 20	(39,264) (41,258) (45,0)	(39,264) (43,6) (45,0)	(39,1614) (41,394) (47,264) (49,0)
40 x 40	(79,528) (81,352) (91,156) (97,0)	(79,528) (81,352) (91,156) (97,0)	(95,1637) (115,880)
60 x 60	(119,243) (123,33) (127,0)	(119,243) (123,33) (127,0)	(159,4761) (165,1979) (167,165) (233,99)
80 x 80	(159,129) (161,69) (163,0)	(159,129) (161,69) (163,0)	(223,2327) (291,1627)
100 x 100	(199,30) (201,6) (203,0)	(199,30) (201,6) (203,0)	(311,4827) (327,3042) (341,1989) (353,45) (365,0)
120 x 120	(239,77) (241,44)	(239,77) (241,44) (261,0)	(363,11340) (413,8821) (417,4613) (455,3830) (517,1292)
140 x 140	(210,1774) (212,1728) (214,40) (244,0)	(210,1774) (212,1344) (214,40) (244,0)	(306,16134) (336,10836) (368,6510) (390,4876) (500,3335) (514,2328)
160 x 160	(319,913) (321,581) (323,0)	(319,601) (321,354) (323,0)	(497,41016) (563,19675) (719,14811) (859,9448)

ary based algorithms is its adaptivity to partially observable dynamic environments. To show this advantage, new tests are done with randomized maps of sizes 60 x 60, 80 x 80, 100 x 100 and 120 x 120. 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 10% to 60% and execution times were observed. An example search space of MOD* Lite with 30% sensor range can be seen in Figure 7. 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 $(18, 230)$, 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 8, 9, 10 and 11. 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 a better path with respect to MOA* or MOGPP or vice versa.

As threat zones and their risk values are used as the second objective, percentages of these zones also affect execution time and generated path quality. In this set of tests different threat zone percents are tested on a fully observable 60 x 60

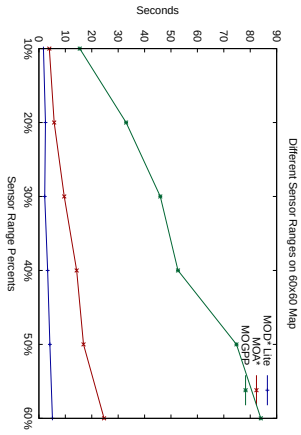


Fig. 8. 60x60 Partially Observable Map on Different Sensor Ranges

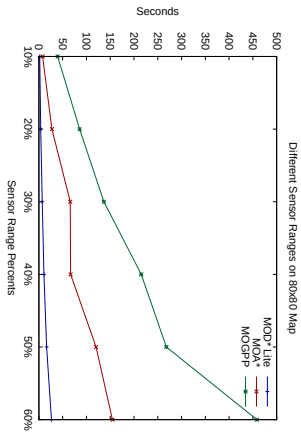


Fig. 9. 80x80 Partially Observable Map on Different Sensor Ranges

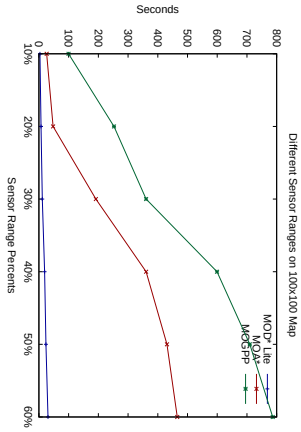


Fig. 10. 100x100 Partially Observable Map on Different Sensor Ranges

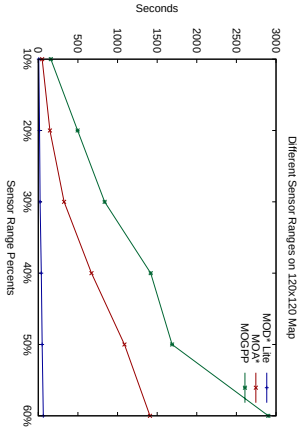


Fig. 11. 120x120 Partially Observable Map on Different Sensor Ranges

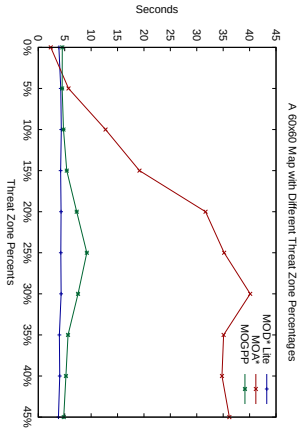


Fig. 12. Execution times of 60x60 Fully Observable Map on Different Threat Zone Percents

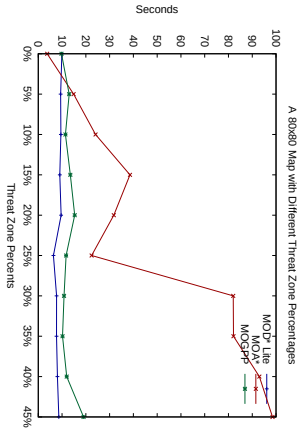


Fig. 13. Execution times of 80x80 Fully Observable Map on Different Threat Zone Percents

TABLE III
PATH COSTS FOR PARTIALLY OBSERVABLE MAPS

Map Size	Frame Size(%)	MOD* Lite	MOA*	MOGPP
60 x 60	10	(126,124)	(132,0)	(136,248)
	20	(124,124)	(120,0)	(148,18)
	30	(118,0)	(118,0)	(142,0)
	40	(118,0)	(118,0)	(144,0)
	50	(118,0)	(118,0)	(168,0)
	60	(118,0)	(118,0)	(178,0)
80 x 80	10	(170,0)	(158,0)	(194,116)
	20	(162,0)	(158,0)	(186,283)
	30	(162,0)	(158,0)	(252,142)
	40	(162,0)	(166,0)	(258,259)
	50	(162,0)	(158,0)	(266,233)
	60	(162,0)	(158,0)	(240,115)
100 x 100	10	(198,0)	(198,0)	(254,715)
	20	(198,0)	(208,0)	(286,896)
	30	(198,0)	(198,0)	(294,1305)
	40	(198,0)	(198,0)	(398,2295)
	50	(198,0)	(198,0)	(326,467)
	60	(198,0)	(198,0)	(370,928)
120 x 120	10	(238,0)	(238,0)	(324,2141)
	20	(238,0)	(238,0)	(382,124)
	30	(238,0)	(238,0)	(464,0)
	40	(238,0)	(238,0)	(448,0)
	50	(238,0)	(238,0)	(542,0)
	60	(238,0)	(238,0)	(482,62)

and 80 x 80 maps and results are given in Figure 12 and 13, respectively. It could be observed that increasing risks of threat zones does not affect performances 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.

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a new method, MOD* Lite, for searching, planning and finding paths on known or 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 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 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 complex applications. To our best knowledge, MOD* Lite is the only incremental search method that can be used for optimizing multiple objectives in dynamic and partially observable search domains, which is classified as the hardest class of problems in state space search.

There exists several incremental moving target solutions for virtual environment proposed in recent years [24], [25], [26]. As a future research, we will attempt to extend our work for non-stationary targets. Also MOD* Lite could be reconsidered in multi agent perspective where each agent distributively execute its planner and cooperate with others to reach a target location.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs.," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

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

[3] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 23, pp. 189 – 211, 1990.

[4] A. T. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, vol. 4, pp. 3310 – 3317, May 1994.

[5] S. Koenig and M. Likhachev, "D*lite," in *Eighteenth national conference on Artificial intelligence*, (Menlo Park, CA, USA), pp. 476–483, American Association for Artificial Intelligence, 2002.

[6] A. Stentz, "The focussed d* algorithm for real-time replanning," in *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2*, IJCAI'95, (San Francisco, CA, USA), pp. 1652–1659, Morgan Kaufmann Publishers Inc., 1995.

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

[8] J. Pangilinan and G. Janssens, "Evolutionary algorithms for the multiobjective shortest path problem," vol. 21 (PWASET), pp. 205–210.

[9] T. Oral and F. Polat, "A multi-objective incremental path planning algorithm for mobile agents," in *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2012 IEEE/WIC/ACM International Conferences on*, vol. 2, pp. 401–408, Dec 2012.

[10] B. S. Stewart and C. C. White, III, "Multiobjective a*," *J. ACM*, vol. 38, pp. 775–814, Oct. 1991.

[11] S. Bayili and F. Polat, "Limited-damage a*: A path search algorithm that considers damage as a feasibility criterion," *Knowledge-Based Systems*, vol. 24, no. 4, pp. 501 – 512, 2011.

[12] F. Guo, H. Wang, and Y. Tian, "Multi-objective path planning for unrestricted mobile," in *Automation and Logistics, 2009. ICAL '09. IEEE International Conference*, pp. 1046 –1051, 2009.

[13] I. Mitchell and S. Sastry, "Continuous path planning with multiple constraints," in *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, vol. 5, pp. 5502 – 5507 Vol.5, 2003.

[14] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," tech. rep., 2001.

[15] O. Castillo, L. Trujillo, and P. Melin, "Multiple Objective Genetic Algorithms for Path-planning Optimization in Autonomous Mobile Robots," *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, vol. 11, pp. 269–279–279, Feb. 2007.

[16] N. Bukhari, A. Khan, A. R. Baig, and K. Zafar, "Optimization of route planning using simulated ant agent system," *International Journal of Computer Applications*, vol. 4, pp. 1–4, August 2010. Published by Foundation of Computer Science.

[17] A. Z. Nasrollahy and H. H. S. 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*, vol. 0, pp. 60–65, 2009.

[18] 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*, pp. 684 –689, May 1998.

[19] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. J.Wiley & Sons, 1st ed., June 2001.

[20] C. A. Coello, "An updated survey of ga-based multiobjective optimization techniques," *ACM Comput. Surv.*, vol. 32, pp. 109–143, June 2000.

[21] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artif. Intell.*, vol. 155, pp. 93–146, May 2004.

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

[23] J. L. Foo, J. Knutzon, V. Kalivarapu, J. Oliver, and E. 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*, vol. 6, no. April, pp. 271–290, 2009.

[24] X. Sun, W. Yeoh, and S. Koenig, "Efficient incremental search for moving target search," in *Proceedings of the 21st international jont conference on Artifical intelligence*, (San Francisco, CA, USA), pp. 615–620, Morgan Kaufmann Publishers Inc., 2009.

[25] X. Sun, W. Yeoh, and S. 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, (Richland, SC), pp. 1081–1088, International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[26] X. Sun, W. Yeoh, and S. Koenig, "Moving target d* lite," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, (Richland, SC), pp. 67–74, International Foundation for Autonomous Agents and Multiagent Systems, 2010.