A Multi-Objective Incremental Path Planning Algorithm for Mobile Agents

Tuğcem Oral
Department of Computer Engineering
Middle East Technical University
06531, Ankara, Turkey
tugcem.oral@gmail.com

Faruk Polat
Department of Computer Engineering
Middle East Technical University
06531, Ankara, Turkey
polat@ceng.metu.edu.tr

***Abstract*—Path planning is a crucial issue in unknown environments where an autonomous mobile agent has to reach a particular destination from some initial location. There are several incremental algorithms such as D* [1], D* Lite [2] that are able to ensure reasonable paths in terms of path length in unknown environments. However, in many real-world problems we realize that path length is not only the sole objective. For example in computer games, a non-player character needs to not only find a minimum cost path to some target location but also minimize threat exposure. This means that path planning/finding activity of an agent in a multi-agent environment has to consider more than one objective to be achieved. In this paper, we propose a new incremental search algorithm called MOD* Lite extending Koenig's D* Lite algorithm and show that MOD* Lite is able to optimize path quality in more than one criteria that cannot be transformed to each other. Experimental results show that MOD* Lite is able to find optimal solutions and is fast enough to be used in real-world multi-agent applications such as robotics, computer games, or virtual simulations.**

***Index Terms*—Path planning; multi-objectivity; incremental search algorithm**

I. 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-life applications including robotics, virtual simulations, computer games, etc. The path planning algorithms can be classified into three categories: off-line algorithms [3] [4], on-line algorithms [5] and incremental algorithms [1], [2], [6]. Off-line path planning algorithms find the whole solution in advance before starting 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 move. 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.

Existing incremental algorithms for path planning problem attempt to minimize path length. However, in many real-life 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 most 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, but possibly the most dangerous one. And also the safest one 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 safety metric, and vice versa. This requirement raises the problem of handling multiple criteria decision making. In this paper we introduce an incremental path finding algorithm, called Multi-Objective D* Lite (MOD* Lite) which extends D* Lite [2], an existing incremental algorithm. MOD* Lite can be used in the design of an autonomous mobile agent facing with the problem of navigation in a partially known environment that need to optimize a predefined set of independent objectives (criteria). Agent can 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 solutions we compared MOD* Lite against MOA* algorithm [7], an offline algorithm, on some test environments that are fully observable. We also tested the performance of MOD* Lite on several partially observable environments guaranteeing the optimal solutions but outperforming the MOA* version modified for unknown environments.

The organization of the paper is as follows: Section II includes the background and related work on path planning. The problem definition and the characteristics of the environment are given in Section III. MOD* Lite and its details are given in Section IV. Section V presents test cases and experimental results. Concluding remarks and future research directions are provided in Section VI.

II. Related Work

In the literature, there are several algorithms focusing on path planning. In this section we would like to introduce existing research relevant to our study where the motivation is to handle existence of multiple objectives and partial observability in path planning. In that respect, we will highlight only multi-objectivity and incremental search here.

Multi-objective A* (MOA*) [7] extends classical A* [4] to handle multiple objectives 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 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* and *CLOSED* sets. Non-dominated states are maintained in a subset of *OPEN* named *ND*. At each iteration of the algorithm, the best alternative is selected from *ND* and expanded until it becomes empty. Finally, solution paths are generated by following back-pointers from goal to start.

Bayili and Polat introduced a multi-objective path planning algorithm, Limited Damage A* [8] 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 discontinue 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 [9]. 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 [10]. The goal of MOPP was to obtain the safe -without colliding 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 compared with single-objective A* algorithm.

Evolutionary methods were also proposed for multi-objective path planning. A recent study by Pangilinan et al. [11] has introduced an evolutionary algorithm for multi-objective shortest path problem and their results show that the 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 [12]. They defined a genetic offline point-to-point agent path planner works on 2-D static grid environment. Bukhari et al. came up with an optimization technique for dynamic on-line path planning in [13] using particle swarm optimization. The proposed algorithm generated and optimized paths in complex and large environments with several constraints. Complete discussion of multi-objective evolutionary algorithms (MOEA) can be found in [14].

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* [1] which guarantees to be optimal and is functionally equivalent to re-planning from scratch. Later, D* Lite was proposed by Koenig et. al [2] which utilized the same navigation strategy with D* but algorithmically different. It was based on Lifelong Planning A* (LPA*) [15]. 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*.

III. Problem Definition & Environment

A. Definition of the Problem

Consider that an unmanned aerial vehicle (UAV) is landing 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 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 shoot the target but the risk of being hit by an air defense unit must be minimized. This means that the UAV should both find a shortest and safest path *as fast as possible*.

In this real-world problem, the UAV has to execute a planner and quickly find available paths. Also it must re-plan its path quickly when a part of the environment become known as it navigates. Evolutionary search algorithms focus on this issue and come up with several solutions [16], [17]. It is obvious that existing algorithms are insufficient for reflecting 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 some part of the environment becomes known.

Our algorithm 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. For the sake of simplicity we restrict ourselves to the UAV domain we have just introduced above and presume the existing of two objectives to be minimized, namely the distance and the degree of danger.

B. Environmental Properties

We illustrate MOD* Lite applied to the UAV path finding task in a 2-D grid based environment. It is easier to present the algorithm and also demonstrate its effectiveness on a such 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 him/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 we presume that agent has enough memory space to maintain the 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 crossed over 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 agent has to enter a threat zone, it prefers to pass through outer levels. With threat zones, agent must both think about the shortest and the safest path. The environment has randomly placed different sized threat zones.

IV. MOD* LITE

A. Overview

Multi-objective problems focus on considering more than one objective at the same time. Consequently, *all* scalar values and atomic operations (like addition, checking for equality, etc.) are to be adapted into vectors of scalars. This causes cost functions 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 $u = v$ we have a fourth alternative meaning that u and v cannot be compared. u and v are said to be equal if the corresponding objective values are equal. We say that u *dominates* v if u is better in at least one objective compared to v. 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.

B. 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 one-step-lookahead values of $g(s)$, the $rhs(s)$ functions are inherited from D* Lite. However, as we have to consider more than one objective, these functions are to return vectors 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 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-dominating 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. In case we have 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 we may have several vectors for any state. In case we need to compare whether one state is better than another, we need to compare two sets of vectors, as formulated below.

Definition We say that u *completely dominates* v iff $\forall x \in u$ $\forall y \in v$ x dominates y. Assume that $v_1 = \{[2, 5], [3, 4]\}, v_2 = \{[6, 1], [5, 2]\}, v_3 = \{[3, 5]\}$ are lists of objective vectors for three states. v_1 and v_2 are non-dominated whereas v_1 completely dominates v_3.

We could frankly say 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 or *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 resides on more than

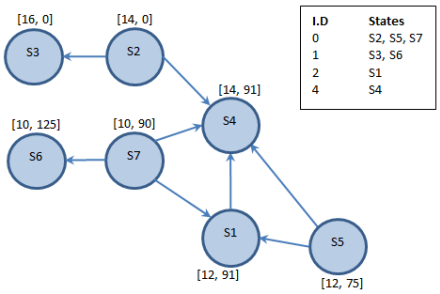

Fig. 1. Directed Acyclic State Expansion Graph

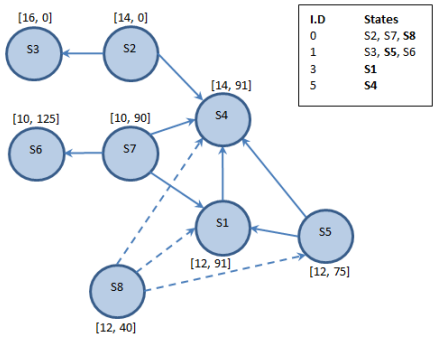

Fig. 2. State Expansion Graph after Adding State $S8$

one solution because it can be understood that two or more predecessors of s are non-dominated to each other.

Other multi-objective operations we use are introduced in following subsections. The overall flow of MOD* Lite is given in Algorithm 1.

Algorithm 1 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()
```

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, we come up with a more convenient structure, a directed acyclic state expansion graph instead priority queue which uses topological ordering of states with respect to the key domination. In this model, the graph (U) contains set of nodes each represented with 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 *dominated by* s and the number of nodes that s *dominates*, respectively. The node(s) 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. 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.

C. Key Formulation

In the previous subsection, we state 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 we consider the algorithm 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 $rhs(s)$, $k_2(s) = nonDom(g(s), rhs(s))$. 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 1. k_m is used for heap reordering as defined in D* Lite.

Algorithm 2 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)
```

D. 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 1, 2 and 3.

Searching order of MOD* Lite is from goal to start state, like D* Lite. The main function of MOD* Lite first calls $initialize()$ to setup the execution in Algorithm 1. 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 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 2 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 weight 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 2 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 finalized and corresponding $g(s)$ values are set, multi-objective paths are generated via these $g(s)$ values by given pseudocode in Algorithm 3. Path generation is achieved in two phases: setting parent(s) for each non-dominated successor of expanding state and constructing paths by following 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 has initially have s_{start}. Thus, starting from s_{start}, the while loop iterates until this queue becomes empty. Finding a goal state is not considered on 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 $\{7$-$40\}$.

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' 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 is 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 add. We will elaborate this idea 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.

Algorithm 3 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,
18:                        s'.parents(s")) then
19:                            s'.parents().remove(s")
20:                            s'.parents().put(s, cumulativeC)
21:                        else
22:                            for all cC ∈ cumulativeC do
23:                                for all eC ∈ s'.parents(s'') do
24:                                    if eC.equals(cC) OR
25:                                    eC.dominates(cC) then
26:                                        cumulativeC.remove(cC)
27:                                        break
28:                                    else if cC.dominates(eC) then
29:                                        s'.parents(s").remove(eC)
30:                                        break
31:                                if s'.parents(s") = null then
32:                                    s'.parents().remove(s")
33:                            if !cumulativeC = null then
34:                                s'.parents().put(s, cumulativeC)
35:            if s'.parents.contains(s) AND
36:            !expandingStates.contains(s') then
37:                expandingStates.add(s')
38:    solutionPaths = construct paths recursively traversing parents
39:    return solutionPaths
```

Else, each existing parent of s', say s'' should be compared with the cumulative cost. These operations are shown in lines between {13-34}. 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 add as a parent to s'. Otherwise, if cumulative cost completely dominates s'', it can be inferred that one can reach to 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 can 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 {31-32}) and s is added as a parent if cumulative costs still have non-dominated cost (lines {33-34}).

After organizing parents of s', it is decided to expand it in next iterations. If s is successfully added as a parent and expanding states queue already does not have it, s' is added to the tail of the queue. This can be seen in lines {35-37}.

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.

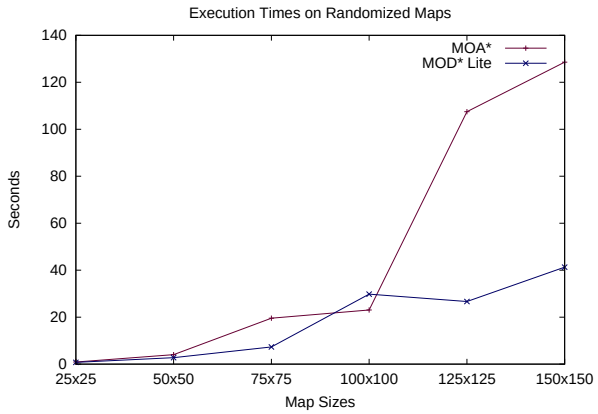

Fig. 3. Execution Times of Randomly Generated Fully Observable Maps

V. Experimental Results

MOD* Lite algorithm is tested on various environments with different scenarios and compared to MOA* that guarantees optimal solutions in fully known environments. First, our aim is to show that MOD* Lite is complete and optimal in fully observable environments. The performance comparison is done in two dimensions, execution times and paths they generate (path quality), respectively. Both algorithms are implemented in Java and run under Linux environment which has Intel Core2 Quad CPU with 2.33GHz and 4 GB RAM.

We generate partially and fully observable 2-D grid environments as detailed in subsection III-B in our tests. The agent tries to find available non-dominated best paths using 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 in all environments used in tests. In the first set of tests, randomly generated 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. In this test group, maps are taken as fully observable. Each of these maps have nearly 30%-32% threat zone and 14%-16% obstacle ratio. Agent's initial and target locations are also taken randomly. For this case, execution times and generated paths' costs of different sized maps are given in Figure 3 and Table I. As seen from results, MOD* Lite finds optimal and sub-optimal results while a gradually increases on time manner. Notice that taken risk values depend on environmental properties and should not be compared between different size of maps.

In the second set of tests, we construct handcrafted maps with same sizes, threat zone and obstacle ratio with randomized tests as indicated above. These maps are also assumed fully observable and agent's initial and target locations are

TABLE I
FOUND NON-DOMINATED PATH COSTS FOR RANDOMIZED MAPS

Map Size	MOD* Lite	MOA*
25 x 25	(49, 571)	(49, 571)
	(51, 10)	(51, 10)
50 x 50	(99, 982)	(99, 982)
	(101, 0)	(101, 0)
75 x 75	(149, 1549)	(149, 115)
	(151, 221)	(153, 0)
	(153, 0)	
100 x 100	(199, 10)	(199, 10)
	(201, 4)	(201, 4)
	(203, 0)	(203, 0)
125 x 125	(90, 1036)	(90, 1036)
	(92, 293)	(92, 293)
	(94, 165)	(94, 165)
	(96, 101)	(96, 101)
150 x 150	(126, 128)	(126, 128)

Fig. 4. Execution Times of Handcrafted Fully Observable Maps

taken randomly. All handcrafted test environments *guarantee* that at least two non-dominated paths will be available. Execution times are shown in Figure 4 and generated paths' costs are given in Table II. In both set of tests, it can be seen that MOA* consumes exponentially increased times to find optimal solutions especially when map sizes increase.

As we use threat zones and their risk values as the second objective, percentages of these zones also effect execution time

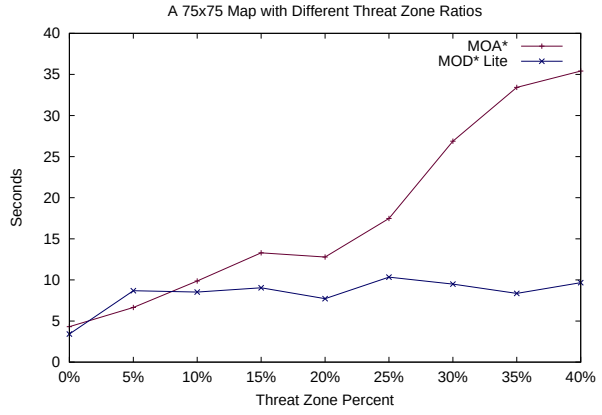

Fig. 5. 75x75 Fully Observable Map on Different Threat Zones

TABLE II
FOUND NON-DOMINATED PATH COSTS FOR HANDCRAFTED MAPS

Map Size	MOD* Lite	MOA*
25 x 25	(49, 722)	(49, 722)
	(51, 505)	(51, 505)
	(55, 216)	(55, 216)
50 x 50	(99, 14)	(99, 14)
	(101, 0)	(101, 0)
75 x 75	(149, 1927)	(149, 1927)
	(151, 1329)	(151, 1329)
	(153, 279)	(153, 279)
	(159, 0)	(159, 0)
100 x 100	(199, 1077)	(199, 144)
	(201, 20)	(201, 20)
	(205, 0)	(205, 0)
125 x 125	(249, 15)	(249, 15)
		(257, 0)
150 x 150	(301, 20)	(299, 145)
		(301, 20)
		(315, 0)

and generated path quality. Different threat zone percents are also tested on a fully observable 75 x 75 map and results are given in Figure 5. We could say that increasing risks of threat zones does not effect performance of MOD* Lite much, it finds results on approximately near times. However, MOA* tightly coupled with it and execution time increases gradually as the threat percentage increases.

In fourth set of tests, we generate partially observable randomized maps of sizes 75 x 75, 100 x 100 and 125x125. On these maps, agent's initial and target locations are chosen to be the furthermost cells in the environment. For each map, we set agent' s sensor range from 20% to 60% and observe execution times. In these tests, the agent starts to plan a path towards the nearest available cell within its sensor range -the temporary target- to the actual target 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 strategy could be set explicitly according to the domain that 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 target and re-executes path planner algorithm. This process iterates until the agent reaches to the desired target 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* to work on exponentially long times. Total execution times to reach to the target for these test cases are given in Figure 6. As can be seen from results, MOD* Lite can easily handle the dynamical issues of the environment where MOA* fails. Due to discovering different parts of the environment during execution, actual traversed path's costs of MOD* Lite and MOA* might be slightly different from each other, where MOD* Lite could follow a

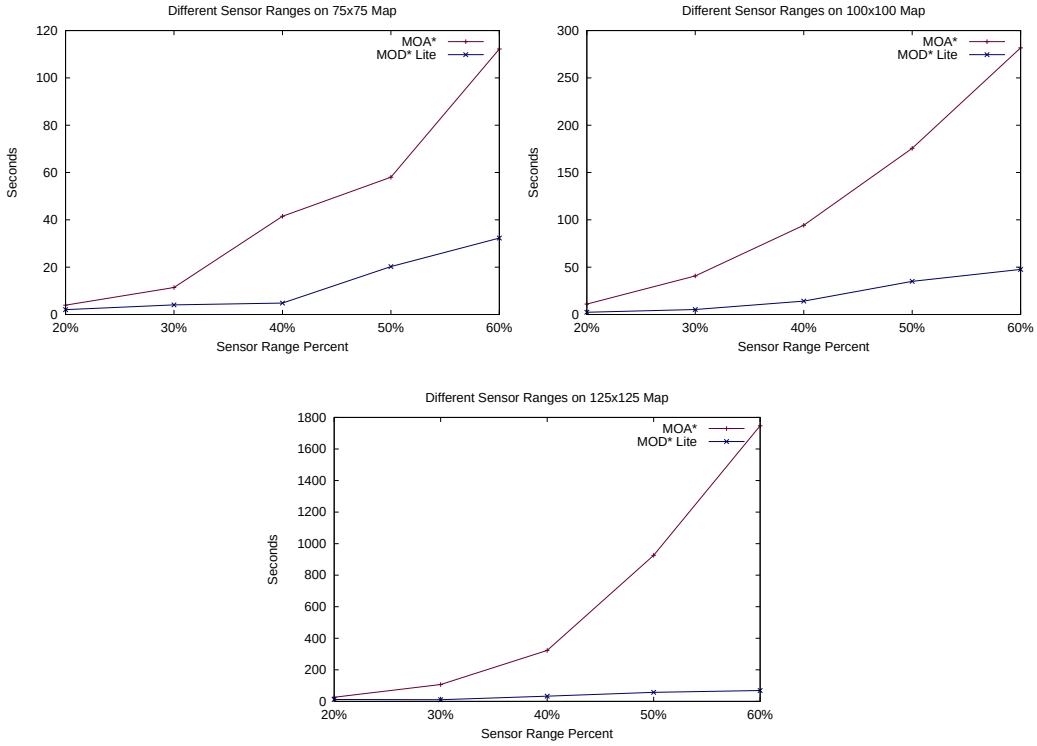

Fig. 6. Sensor Ranges on Different Maps

better path with respect to MOA* or vice versa.

VI. Conclusions & Future Work

In this paper, we present MOD* Lite, a novel approach for searching, planning and finding paths on known and unknown 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 brings multi-objectivity to the solution space successfully, which is required in many real-world problems. Experimental results show that MOD* Lite is able to optimize path quality and is fast enough to be used in real-world multi-agent applications such as robotics, computer games, or virtual simulations. To our best knowledge, MOD* Lite is the first incremental search algorithm to handle multi-objectivity. We are planning to extend MOD* Lite for moving targets.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] K. Deb and D. Kalyanmoy, *Multi-Objective Optimization Using Evolutionary Algorithms*. J.Wiley & Sons, 1 ed., June 2001.

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

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

[17] 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.