Here is the half-coded half-pseudocoded MOD*Lite algorithm extends from D*Lite:

```
/**
 * Calculates the summation of all objectives in given objective array lists.
 * Notice that given arrays are additive that they have same objective sizes and same objective
 * behaviours.
 */
function sum(ObjectiveArray oa1, ObjectiveArray o2) {
        // create an objective array for summation.
        ObjectiveArray summation;
        for each objective in oa1 and oa2{
                summedValue = oa1.get() + oa2.get()
                summation.put(summedValue);
        }
}

/**
 * Given objective array lists are assumed to contain non-dominated objective arrays, so
 * this function compare each objective array with the one on the other list and finds fully-non-
 * dominated list of objective arrays.
 */
function nonDominatedList(List<ObjectiveArray> list1, List<ObjectiveArray> list2){
        // traverse two lists.
        for each objective array in list1 {
            for each objective array in list2 {
                if (oa1 dominates oa2)
                    list2.remove(oa2)
                else if (oa2 dominates oa1)
                    list1.remove(oa1)
                // if none of above conditions are true,
                // means that none of objective arrays dominate each other.
            }
        }

        // return all remaining items in these lists.
        return list1.addAll(list2);
}

/**
 * Calculates keys as the same way in D*Lite, but with several objectives.
 */
function calculateKey(State s){
        // multiobjective sum of (h(start) + s) + km
        k1 = sum(sum(h(start), s), km);
        // get non-dominated list of g(s) and rhs(s)
        k2 = nonDominatedList(g.get(s), rhs.get(s));
        return [k1, k2];
}
```

```
/**
 * This function initializes the constraints of MOD*Lite
**/
function initialize() {
	// objective behaviours are assumed to be both minimized
	ObjectiveBehaviours = [MINIMIZED, MINIMIZED];
	// create a 2D grid world map including obstacles and threat zones.
	initializeMap(objectiveBehaviours);
	// start and goal coordinates for initial location of agent.
	// WIDTH and HEIGHT are the boundaries of the world, respectively.
	// These boundaries are parameterized.
	start = [0, 0];
	goal = [WIDTH - 1, HEIGHT - 1];
	// Priority queue is empty
	U = Empty
	// ZERO Objective Array is [0 (MIN), 0(MIN)]
	km = ObjectiveArray.ZERO;

	rhs.put(goal, ObjectiveArray.ZERO);
	U.insert(goal, calculateKey(goal));
}

/**
 * Updates the given state u' s rhs value. Also re-inserts it into priority-queue (U) with a new
 * calculated key value.
 */
function updateVertex(State u) {
	// min of c(u, s' + g(s')) is calculated between the successors of u.
	// While calculation, if there is a non-domination criteria between two successors of u,
	// the non-dominated list is generated.
	if (u not equals goal) rhs.put(u, min_{s' in successor(u)}(sum( c(u, s'), g.get(s'))));
	if (U.contains(u)) U.remove(u);
	// actually, this equality search is a multi-objective equality because which means
	// all objective arrays are equal to each other.
	// Notice that g and rhs holds lists of objective arrays
	if (g.get(u) equals rhs.get(u)) U.insert(u, calculateKey(u));
}
```

```
/**
 * Re-orders the priority-queue with respect to states' keys and updates g values.
 */
function computeShortestPath() {
	loopCount = 0;
	// We should find a more robust strategy to finalize this loop, this counter is used because
	// priority-queue' s compareTo method should not always returns decidable results. For
	// instance; if k1 completely dominates k2, we could say that k1 is smaller so it returns -1. Or
	// vice-versa, it we could say k2 is smaller so it returns 1. But when these keys are
	// non-dominated to each other, it returns 0 and according to this action, we might insert
	// corresponding state into wrong position in the queue, so this loop should never end.
	while( U is not empty AND
		(U.topKey().compareTo(calculateKey(start)) < 0) OR (not equals(rhs(start), g(start)))
		AND loopCount < 5000/*or a predefined constant value*/){

		kOld = U.topkey();
		u = U.pop();

		if (kOld.compareTo(calculateKey(u)) < 0) U.insert(u, calculateKey(u));
		else if ( rhs(u) completely dominates g(u)) {
			g(u) = rhs(u);
			for each predecessor(u) : s
			    updateVertex(s);
		} else {
			g(u) = INFINITY_FOR_MINIMIZED_BEHAVIOUR;
			for each predecessor(u) : s
			    updateVertex(s);
			updateVertex(u);
		}
		loopCount++;
	}
}
```

```
/**
 * Recomputes the lowest cost path through the map, taking into account any
 * changes in start location and edge costs. If no path can be found this
 * will return an empty list.
 *
 * @return a list of states from start to goal
 */
function plan() {

	// This map is used to hold states and paths (from start) to reach them. Once
	// a state is expanded, it is removed from this map. Eventually, only the goal state resides in
	// this map with its all potential paths from start.
	Map nonDominatedPaths = [];

	// minimum states list – the states to be expanded.
	List minStates = [];
	minStates.add(start);

	initialize();
	computeShortestPath();
	// according to our requirements, while loop is switched with for; to expand nodes.
	for each state s in minStates{
		// if we reach goal state, expand remaining nodes.
		if (s.equals(goal)) continue;
		// If g(start) = Inf, then there is no known path - for this s, continue on other states.
		if (g(start) is INFINITY_FOR_MINIMIZED_BEHAVIOUR) continue;

		for each state s' in successors of s {
			find minimum c(s, s') + g(s');
			if non-domination case occurs, add s' to potential non-dominated successors list;
		}

		for each state s' in non-dominated successors list {
			construct a path from s to s';
			add this path to nonDominatedPaths with [s', path];
		}
		remove s from nonDominatedPaths;
		add all potential non-dominated successors to minStates to expand next.
	}

	return nonDominatedPaths.get(goal);
}
```