A Randomized Heuristic Algorithm for Cyclic Routing of UAVs

Cheng Siang Lim and Shell Ying Huang
School of Computer Science and Engineering
Nanyang Technological University
Singapore
LIMC0183@e.ntu.edu.sg, assyhuang@ntu.edu.sg

***Abstract*—Unmanned Aerial Vehicles (UAVs) have been increasingly used in military and civilian applications. Even though the UAV routing problems have similarities with Vehicle Routing Problems, there are still many problems where effective and efficient solutions are lacking. We propose a randomized heuristic algorithm for the cyclic routing of a single UAV. The UAV is required to visit a set of target areas where the time interval between consecutive visits to each area cannot exceed its relative deadline. The PSPACE-complete problem has a solution whose length may be exponential. Our algorithm tries to compute a feasible cyclic route while trying to keep short the cycle time. Our tests of 57 instances of the problem show that the algorithm has good effectiveness and efficiency.**

Keywords—Single Unmanned Aerial Vehicle (UAV); cyclic routing; randomization; heuristic

I. INTRODUCTION

In recent years the deployment of Unmanned Aerial Vehicles (UAVs) for various purposes has been increasing. UAVs are used for surveillance and monitoring, mobile target tracking, search and rescue, delivery of small parcels and taking videos or photographs which are otherwise difficult to obtain from ground level. In this paper, we focus on the problem of cyclic routing of a single UAV for the purpose of surveillance and monitoring. There is a set of target areas to be continuously monitored. Each of the target areas needs to be visited repeatedly. The time interval between consecutive visits by the UAV to an area cannot exceed its relative deadline. Since the problem is PSPACE-complete [8], we propose a randomized heuristic to solve it. The objective of the algorithm is to find a cyclic route for the UAV that satisfies all the relative deadlines and minimizes the total distance traveled.

The rest of the paper is organized as follows. We review related work on UAVs in Section II. The cyclic routing problem and our algorithm are presented in Section III. Then Section IV presents the tests we have done to evaluate the effectiveness and efficiency of our algorithm. Section V gives the conclusion.

II. RELATED WORK

The UAV routing problem has been studied mainly from two levels. At the lower level, it is about guidance, navigation and control. Examples of the published works include the survey of these technologies by Kendoul [9], the navigation and control of the AR drone by Bristeau et al. [3], two complete system architectures by Elkaim et al. [5]. At the higher level, it is about routing.

A number of studies have been done to route multiple UAVs to multiple target areas. Pohl and Lamont [12] proposed a multi-objective optimization algorithm for routing multiple UAVs to multiple locations using genetic algorithm approach. In their problem, each location has a time window for an UAV to visit. The objective is to minimize the number of UAVs, the waiting time of UAVs (if an UAV arrives before the start of the time window of a location) and the total travelling distance. Enright et al. [6] presented a survey on the algorithms for assigning and scheduling of one or more UAVs to perform tasks which dynamically appear in various locations. The objective of the algorithms is to minimize the average time between the times of appearances of the tasks and their completion times. Mersheeva and Friedrich [10] proposed a heuristic method for multi-UAV routing with priorities and limited energy/power. The UAVs can be used to monitor crime scenes or disaster sites. The objective is to maximize the number of sites with up-to-date information under the constraint of limited energy resources. Park et al. [11] proposed a 2-phase heuristics to route an unmanned combat vehicle to patrol a set of checkpoints where the probability of enemy infiltration to each checkpoint increases nonlinearly with time.

For the cyclic routing of UAVs, Basilico et al. [1], [2] formulated the problem as a constraint satisfaction problem and proposed a solution method to find a feasible cyclic route for a single UAV. The algorithm searches the solution space by backtracking. To improve the efficiency a forward checking method is used to reduce the branching of the search tree. However, this solution is believed to be wrong [4] since it is based on the theorem that there is a polynomial bound on the length of the cyclic route, which implies NP-completeness of the problem. In fact, Ho and Ouaknine [8] proved that the cyclic routing of UAVs is PSPACE-complete, even in the single UAV case. Fargeas et al. [7] also proposed heuristic to compute a cyclic route for a mobile agent to satisfy re-visit rates. Drucker et al. [4] presented a lower bound and an upper bound on the number of UAVs for solving the cyclic routing problem and constraints models for single and multiple UAV problems, respectively. They solved their constraints models by a Satisfiability Modulo Theories (SMT) solver called Z3.

III. ROUTING ALGORITHM

A. Cyclic Routing Problem

Let G = (V, E) be a weighted graph and R = [r_1, r_2, …, r_n] be a vector. V = {v_1, v_2, …, v_n} is the set of target areas to be monitored and E = {f_{ij}}, (i = 1, 2, …, n; j = 1, 2, …, n), is the 2-dimentional matrix specifying the flight times between vertices in V. R is a vector of relative deadlines for the target areas. The relative deadline of an area is the longest time interval allowed between two consecutive visits to the area by the UAV. There is a surveillance time required by the UAV at each visit to a target area.

The assumptions are as follows:

- Target areas can be entered by the UAV from any direction.
- The surveillance time is large enough to allow the UAV to leave a target area from the point that is closest to the next area that it is going to visit. Thus the given flight time between a pair of target areas is the shortest.
- Flight times between areas are constants.
- Flight times and relative deadlines are given in integers as input data.
- For a triangle formed by any three vertices, the triangle inequality theorem is obeyed, i.e. the sum of two sides is greater than the third.
- Each flight time is at most half of the relative deadline of the target that the UAV is traveling to or from. This means there is no isolated vertex which will require another UAV to manage.

Let t_i, i = 1, 2, …, n, be the surveillance time. We can simplify the problem by distributing the surveillance time into flight times such that each f_{ij} in E is added by $(t_i + t_j)/2$. After this simplification, the cyclic routing problem is: given a graph G and a vector R, find a cyclic path of minimum length which starts from a vertex, visits every vertex at least once (including the starting vertex) and returns to the starting vertex while all relative deadline constraints are satisfied.

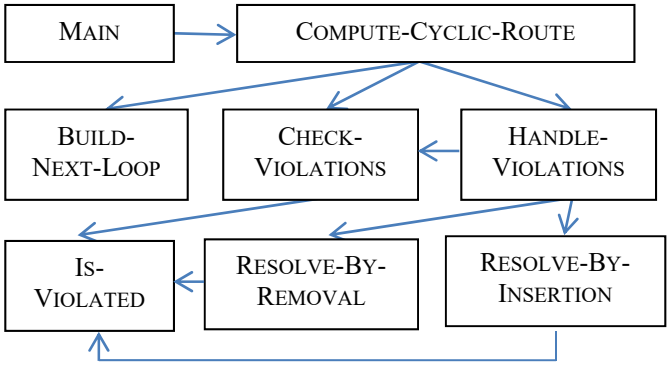

Fig. 1. Main components.

B. Basic Randomized Routing Algorithm

We first present a basic algorithm whose main objective is to find a feasible solution with good total flying distance by the UAV. The main components of the algorithm for the computation of cyclic route are shown in Fig. 1. The name of each component tells the function of that component. The arrows in the figure show the caller-callee relationship between two components. The whole algorithm starts from the MAIN function.

As shown in Fig. 2, the MAIN function repeatedly calls COMPUTE-CYCLIC-ROUTE to find a solution. When one solution is found, it terminates. Otherwise, it continues until 1000 trials have been attempted. Since there are randomized decisions in the construction of a cyclic route, a failure in one attempt does not mean there is no feasible solution. The number of trials can be set by the user of the algorithm. In our experiments, we find that for most problem instances where a solution is found by our algorithm, the number of trials is smaller than 30 (we tested graphs up to 7 vertices).

```
function MAIN()
{
    V ← input vertex set
    R ← input relative deadlines
    E ← input flight times
    trials ← 0
    repeat
        trials ← trials + 1
        solution ← COMPUTE-CYCLIC-ROUTE(V, E, R)
        if solution is not None then return True
    until trials = 1000
    return False
}
```

Fig. 2. Main function.

In the pseudo code for COMPUTE-CYCLIC-ROUTE in Fig. 3, *V* is the set of target areas (vertices), *E* is the matrix of flight times between each pair of vertices, *R* is the array of relative deadlines for the vertices. The data structures used are as follows:

Solution is the cyclic route being constructed which is presented by a vector of vertices together with the time the UAV reaches each of these vertices and whether a vertex at this position in the cyclic route is locked (cannot be removed when handling relative deadline violations). An example of a *solution* (which is the *route* in a number of functions in Fig. 4-9) when *V* = {a, b, c, d}: a→b (5 time units) →c (3 time units) →d (5 time units) →a (3 time units).

position	0	1	2	3	4
vertex	a	b	c	d	a
travelTime	0	5	8	13	16
locked					

Violated is to hold the list of violations of relative deadlines in *solution*, each entry is indexed by the vertex. If there is a violation the entry stores the start and end positions in the cyclic route which is a section of the route whose traveling time violates the relative deadline of the vertex. For example, *V* = {a, b, c, d}, *R* = {16, 8, 14, 8} and *solution* is

position	0	1	2	3	4	5	6
vertex	d	a	b	d	b	c	d
travelTime	0	3	5	8	11	14	16
locked							

In this case, from position 4 to position 2, the interval for b is 10 (>8) and from position 0 to position 6, the interval for c is 16 (>14). So *violated* contains*:*

vertex	b	c
start	4	0
end	2	6

UnvistedSet is the set of vertices which have not been included in the cyclic route *solution*.

FirstVertex is the starting vertex of *solution* which is the starting vertex of every loop in *solution*.

It is observed that the cyclic route for the UAV consists of one or more loops all of which start from a certain vertex *a* and return to *a.* This vertex *a* will be one of the vertices with the minimum relative deadline. For example, a cyclic route involving 4 vertices may just be d → a → c → b → d which means one loop, or d → a → c → d → b → d which means 2 loops. Vertex d will be a vertex with the smallest deadline. So COMPUTE-CYCLIC-ROUTE in Fig. 3 begins with an empty cyclic route, and a loop is built repeatedly to extend the cyclic route. Each time BUILD-NEXT-LOOP will build a loop that includes all current unvisited vertices. After this, CHECK-VIOLATIONS will check for deadline violations and it is followed by HANDLE-VIOLATIONS. The function HANDLE-VIOLATIONS may remove some vertices from the loop just built so that the current cyclic route does not violate any relative deadlines. Then another loop to include these vertices needs to be built in the next iteration of the 'repeat' statement. The iteration stops when all vertices are included in at least one loop in the cyclic route and their relative deadlines are not violated.

Fig. 4 shows the pseudo code for BUILD-NEXT-LOOP. It is a greedy algorithm to build a TSP path including all vertices currently not in the cyclic route. The vertex *firstVertex* is always the starting vertex, which is randomly chosen among the vertices with the shortest relative deadline by COMPUTE-CYCLIC-ROUTE. The first two vertices to be added in the route (not including the starting vertex) are chosen at random among the set of unvisited vertices, as vertices added at this point make little impact to the algorithm. Subsequent vertices are added *in between* existing vertices in the loop or at the end of the route so that it results in the *smallest* increase in cycle time. The function GET-INSERT-POSITION in Fig. 4, whose pseudo code is not presented for brevity, finds the position for inserting a vertex into the loop. Note that each entry in *route* includes the vertex identifier and the time the UAV reaches this vertex since the start time of the route. This is the *travelTime* computed in BUILD-NEXT-LOOP.

```
function COMPUTE-CYCLIC-ROUTE(V, E, R)
{   violated ← { }
    solution ← empty cyclic route
    unvistedSet ← V
    firstVertex ← randomly chosen vertex with the shortest
relative deadline
    repeat
      while violated is not empty do
        if a vertex with the longest deadline is in solution
          then return None
        if not HANDLE-VIOLATIONS(solution, violated,
unvisitedSet)
          then return None
       if unvisitedSet is not empty then
         BUILD-NEXT-LOOP(firstVertex, solution,
unvisitedSet, E)
       CHECK-VIOLATIONS(solution, violated, V, R)
    until violated and unvisitedSet are empty
    return solution
}
```

Fig. 3. Pseudo code for Compute-Cyclic-Route.

```
function BUILD-NEXT-LOOP(firstVertex, route, unvisited,
flightTime)
{ // build a new loop to include all vertices in unvisited
    if route is empty then travelTime ← 0
    else travelTime ← the cycle time of the current route
    route.append(firstVertex, travelTime)
    unvisited.discard(firstVertex)
for  i= 1 to min(2, |unvisited|) do
    lastVertex ← current last vertex in route
    nextVertex ← choose a random vertex in unvisited
    travelTime ← travelTime + flightTime[lastVertex,
nextVertex]
    route.append(nextVertex, travelTime)
    unvisitedSet.discard(nextVertex)
while unvisited is not empty
    nextVertex ← choose a random vertex in unvisited
    start ← the timeslot at which firstVertex is last visited
    end ← no. of vertices in route
    position ← GET-INSERT-POSITION(nextVertex, start,
end)
    route.insert(nextVertex, position)
    unvisitedSet.discard(nextVertex)
}
```

Fig. 4. Pseudo code for BUILD-NEXT-LOOP.

The function CHECK-VIOLATIONS in Fig. 5 checks whether there is a violation of relative deadline in *route* for each vertex in the set of vertices *aList*. The function is called

from several places. When CHECK-VIOLATIONS is called from COMPUTE-CYCLIC-ROUTE, the vertex set V of the whole graph is *aList*. This is because of these two observations: 1) after BUILD-NEXT-LOOP, every vertex that has not been included in previous loop(s) in the cyclic route is included in the current loop; 2) with the inclusion of the current loop, the cycle time of the route is increased so some vertices in previous loop(s) may have their relative deadline violated. Therefore, every vertex needs to be checked. The checking for each vertex is done by IS-VIOLATED shown in Fig. 6.

```
function CHECK-VIOLATIONS(route, violatedRegions, aList, R)
{
  for each vertex v in aList do
      start ← 0
      end ← no. of vertices in route
      if v is visited more than once then
        start ← position in route where v is last visited
        if v ≠ firstVertex then
          end ← position in route where v is first visited
      IS-VIOLATED(v, route, start, end, violatedRegions, R)
}
```

Fig. 5. Pseudo code for CHECK-VIOLATIONS.

When CHECK-VIOLATIONS is called from COMPUTE-CYCLIC-ROUTE, it is after a new loop is built each time. With the addition of the new loop, the cycle time of the cyclic route is increased. So for a vertex which appears only once in the whole cyclic route, function CHECK-VIOLATIONS calls function IS-VIOLATED in Fig. 6 with *start* index and *end* index being the two ends of the cyclic route. This will allow IS-VIOLATED to check whether the cycle time of the whole route exceeds the relative deadline of the vertex. For a vertex which appears more than once (in separate loops) except the starting vertex *firstVertex,* function CHECK-VIOLATIONS calls function IS-VIOLATED with *start* index being the position of the last appearance of the vertex and *end* index being the position of the first appearance of the vertex in the cyclic route. This will allow IS-VIOLATED to check whether the time interval between the last visit and the first visit to the vertex exceeds the relative deadline. There is no need to check the time interval between the ith and (i+1)th visits to the vertex because this check is done when the (i+1)th visit is inserted into the cyclic route. For the starting vertex *firstVertex,* function CHECK-VIOLATIONS calls function IS-VIOLATED with *start* index being the position of the last appearance of the vertex (start point of the new loop) and *end* index being the end of the cyclic route. This will allow IS-VIOLATED to check whether the cycle time of the loop exceeds the relative deadline of the vertex. There is no need to check whether the cycle time of the previous loop(s) violates the relative deadline because it was checked previously when the loop was added to the cyclic route.

With the information passed from CHECK-VIOLATIONS, function IS-VIOLATED in Fig. 6 simply checks whether the time interval between the *start* visit and the *end* visit in the cyclic route exceeds the relative deadline of *vertex*. When a relative deadline is violated, it saves in *violated* the *start* and *end* positions in the cyclic route which is a section of the route whose traveling time violates the relative deadline of the vertex and locks the vertex in the cyclic route to prevent the vertex from being removed by HANDLE-VIOLATIONS in Fig. 7. This is because if the vertex has its relative deadline violated, this vertex should get more visits to shorten the interval between visits and not be removed from the current cyclic route.

```
function IS-VIOLATED(vertex, route, start, end, violated, deadline)
{  if end < start then
     // vertex visited more than once and vertex ≠ firstVertex
        timeInBetween ← the time between vertex is first visited and last visited
        travelTime ← the cycle time − timeInBetween  // This is time-around.
     else if vertex = firstVertex then
        travelTime ← the cycle time − route.times[start]
     else travelTime ← route.times[end] − route.times[start]
     if travelTime > deadline[vertex] then      // is violated.
        violated[vertex] ← (start, end)
        route.locked[position in route where vertex is last visited] ← True    // vertex cannot be removed later
        return True
     if violated ≠ None then violated.discard(vertex)
     return False
}
```

Fig. 6. Pseudo code for IS-VIOLATED.

Function HANDLE-VIOLATIONS in Fig. 7 is to resolve all violations in the current cyclic route as specified by *violatedRegions*. The basic idea of HANDLE-VIOLATIONS in Fig. 7 is that if the *firstVertex* has a violation, it means the cycle time in the last loop in the cyclic route is longer than the relative deadline of the vertex. This violation will be resolved by removing one or more vertices from the loop to reduce the cycle time until it is within the length of the relative deadline. This is done by RESOLVE-BY-REMOVAL.

It was mentioned earlier that if a vertex has a violated relative deadline, it is locked in the route and the vertex will not be removed. So if after RESOLVE-BY-REMOVAL, it is found that the number of vertices with unresolved violations is greater than the number of vertices in *route* (no. of vertices in *route* $\leq |violatedVertices|$), no solution can be found and HANDLE-VIOLATIONS will return false. Function CHECK-VIOLATIONS is called again to check for violations after the removal of vertices.

When a violated vertex is not the *firstVertex*, the violation will need to be resolved by inserting the violated vertex into the cyclic route one more time. For each of such vertices, function RESOLVE-BY-INSERTION is called in a random order to insert it into the cyclic route. There are situations where the violation cannot be resolve by insertion. In such situations, no solution can be found and HANDLE-VIOLATIONS will return false.

```
function HANDLE-VIOLATIONS(route, violatedRegions, unvisitedSet)
{   if firstVertex in violatedRegions then
        if firstVertex is visited only once then
            violatedVertices ← a list of keys in violatedRegions        // Before removal.
        if not RESOLVE-BY-REMOVAL(firstVertex, route, violatedRegions, unvisitedSet) then return False
        if firstVertex is visited only once then
            if no. of vertices in route ≤ |violatedVertices| then return False
        violatedVertices ← a list of keys in violatedRegions
        CHECK-VIOLATIONS(route, violatedRegions, violatedVertices)
    repeat
        violatedVertex ← randomly choose a vertex in violatedRegions
        if not RESOLVE-BY-INSERTION(violatedVertex, route, violatedRegions,unvisitedSet) then return False
    until violatedRegions is empty
    return True
}
```

Fig. 7. Pseudo code for Handle-Violations.

```
function RESOLVE-BY-REMOVAL(violatedVertex, route, violatedRegions, unvistedSet)
{   (start,end) ← violatedRegions[violatedVertex]
    repeat
        if not REMOVE-VERTEX(route, violatedRegions[violatedVertex], unvisitedSet)
            then return False
        end ← violatedRegions[violatedVertex].end
    until not IS-VIOLATED(violatedVertex, route, start, end, violatedRegions)
    violatedRegions.discard(violatedVertex)
    // Vertex is no longer violated.
}
```

Fig. 8. Pseudo code for Resolve-By-Removal.

Function RESOLVE-BY-REMOVAL in Fig. 8 calls function REMOVE-VERTEX (pseudo code not presented for brevity) to remove a vertex from the last loop of cyclic route as demarcated by *start* and *end* values from *violatedRegions*. It will choose a vertex which results in the biggest reduction in cycle time of the last loop. The removed vertex will be put back to *unvisitedSet* so that it can be included in a new loop to be built later. If REMOVE-VERTEX cannot find a vertex to remove, e.g. all vertices in the loop are locked, it will return false to indicate the failure of the resolution of the violation. This will cause HANDLE-VIOLATIONS to return false, i.e. a solution cannot be found. Function IS-VIOLATED is called to check whether the violation of the loop has been resolved after removing a vertex. If that is the case, removal of vertices will stop.

```
function RESOLVE-BY-INSERTION(violatedVertex, route, violatedRegions, unvisitedSet)
{
    timeInBetween ← travel time from the position in route at which violatedVertex is last visited to the last position at which it is presumably inserted
    start ←position of start point of the last loop
    end ← no. of vertices in route
    repeat
        lastVisit ← position in route at which violatedVertex is last visited
        if lastVisit > position of start point of the loop being checked then
            if unvisitedSet not empty then break
            else if travel time from the position in route at which violatedVertex is last visited to one it is first visited ≤ deadline[violatedVertex] then break
            else return False
        position ← GET-INSERT-POSITION(violatedVertex, route, start, end)
        timeInBetween ← travel time from lastVisit to position at which it is presumably inserted
        if timeInBetween ≤ deadline[violatedVertex] then
            // Revisit violatedVertex at position.
            lock ← True
            route.insert(violatedVertex, position, lock)
            end ← end + 1
            violatedRegions.discard(violatedVertex)
            if IS-VIOLATED(firstVertex, route, start, end, violatedRegions) then
                if not Resolve-By-Removal(firstVertex, route, violatedRegions, unvisitedSet) then return False
                position ← position in route at which violatedVertex is last visited
            if INSERT-VIOLATED (position, route)
                then return False
            timeInBetween ← travel time from the position in route at which violatedVertex is last visited to the last position at which it is presumably inserted
            end ← no. of vertices in route
        else // timeInBetween is too long, try previous loop
            end ← start
            if start is the start position of the first loop
                then return False
            start ← start position of the previous loop in route
    until unvisitedSet is not empty and timeInBetween < deadline[violatedVertex]
    violatedRegions.discard(violatedVertex)
    return True
}
```

Fig. 9. Pseudo code for Resolve-By-Insertion.

A vertex needs to be inserted into the cyclic route again if it has been visited before, and the cycle time or time between the

last visit and the first visit is greater than its relative deadline. Function RESOLVE-BY-INSERTION in Fig. 9 will perform this task. The basic idea is to search for a loop to insert the vertex. The search starts from the last loop. Function GET-INSERT-POSITION which is used in BUILD-NEXT-LOOP as mentioned earlier is also used here to find a position in the loop. This will cause minimum increase in cycle time. This tentative position for insertion will be checked to see whether inserting the vertex here will result in an interval between this position and the position of the last visit, i.e. *timeInBetween,* violating the relative deadline of the vertex. If it is the case, search has to continue with the previous loop to find a suitable insertion point. Otherwise the vertex will be inserted. Since this insertion increases the cycle time of the loop, IS-VIOLATED is called to check whether the relative deadline of *firstVertex* is violated. If it is the case, Resolve-By-Removal is called to remove some vertices which will put certain vertex to go into *unVisitedSet*. Function INSERT-VIOLATED (pseudo code not shown here for brevity) is used to check whether the insertion of the vertex will cause an interval between two consecutive visits of any other vertices violating their deadline. If it is the case, return false to indicate failure of vertex insertion.

When trying to continue the search in a previous loop, obviously a vertex cannot be revisited/inserted in the first loop of the cyclic route (condition: *start* is the start position of the first loop). When the condition arises, the attempt to insert is aborted. In addition, there are situations where it is better to *procrastinate* the vertex insertion: when *unvisitedSet* is not empty due to the removal of vertices. Then insertion will not be done for this vertex at this point. The attempt to insert will terminate when *unvisitedSet* is found not empty or after an insertion, the time between the last visit and the first visit of the vertex in *route* is not greater than the relative deadline.

C. Improved Randomized Routing Algorithm

The basic algorithm stops when a solution is found. This solution may be a good solution but a better solution may be found if the randomized basic algorithm is allowed to have more trials.

In an improved version of the algorithm, after a solution is returned, it will continue with the next trial in the repeat loop in Fig. 2. When in the next trial, because of the randomization, a different solution may be returned. The computation of a solution continues until either three solutions with the same (minimum) total cycle time are obtained or (3*the number of vertices with the shortest relative deadline*the total number of vertices) iterations have been tried. Then the solution with the shortest cycle time is returned.

IV. EVALUATION RESULTS

Experiments are conducted to evaluate the effectiveness and the efficiency of the randomized heuristic routing algorithm. The algorithm is implemented in Python 3. Experiments are run on Intel® Core™ i7-6500U CPU @ 2.50 GHz, 8.00 Gb RAM, 64-bit Operating System, x64-based processor.

The following test cases are designed according to 1) the different number of vertices; and 2) the different patterns of solutions that may be found. Different patterns refer to:

- All vertices only need to be visited once (a single loop in the cyclic route solution).
- Only the vertex with the shortest relative deadline needs to be revisited.
- Multiple vertices need to be revisited.
- Various numbers of loops are required in the cyclic route solution.

We tested 6 cases of 3-vertex graphs, 17 cases of 4-vertex graphs, 13 cases of 5-vertex graphs, 12 cases of 6-vertex graphs, 9 cases of 7-vertex graphs. All graphs are complete graphs. Fig. 10 shows the 6 cases of 3-vertex graphs where the number on top of each vertex is the relative deadline. Fig. 11 shows 3 of the 9 cases of 7-vertex graphs tested. The results include 1-loop, 2-loop and 3-loop cyclic routes.

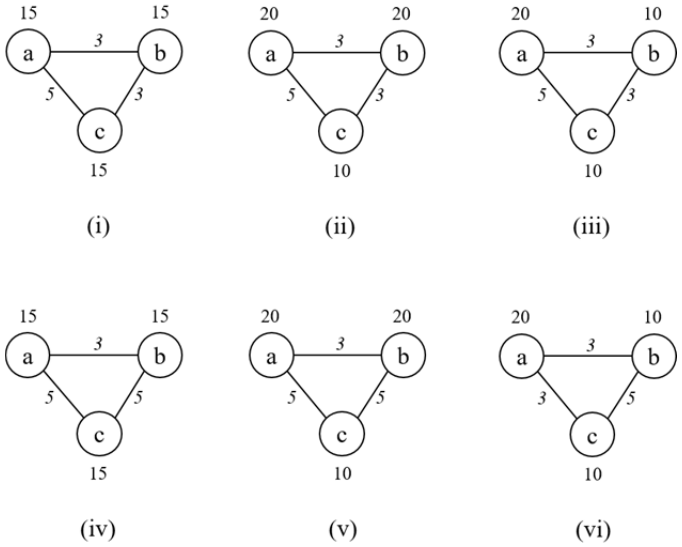

Fig. 10. Six 3-vertex graphs tested.

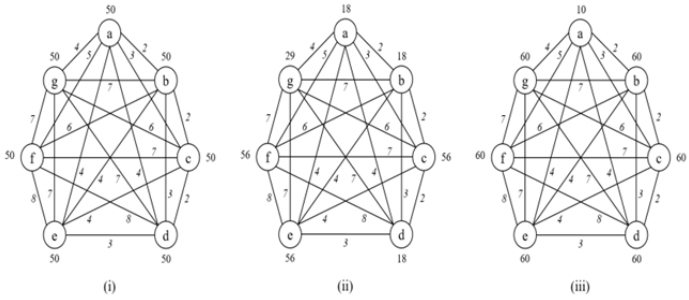

Fig. 11. Three of the nine 7-vertex graphs tested.

Table 1 shows the results of finding a cyclic route for the UAV by the basic algorithm. For each graph, 1000 independent runs are conducted. The graphs are labeled in column 1 in the table, e.g. 7(1) means the first instance of the 7-vertex graph which is shown in the leftmost graph in Fig. 11. The independent runs for each graph may produce cyclic routes of different cycle times because of the randomization in the algorithm. Columns 2 and 3 show the lowest and the highest cycle times obtained among the 1000 runs. When the two

entries for a certain row are blank, it means all 1000 runs produce the same cycle time (may have different cyclic routes). The last two columns show the probabilities of obtaining the lowest and the highest cycle times, respectively, based on the 1000 run results. When these two probabilities do not add up to 100%, it means there are other cycle times (in between the lowest and the highest cycle times) returned by the algorithm.

TABLE I. CYCLE TIMES OF SOLUTIONS FROM BASIC ALGORITHM

case	*Lowest cycle time*	*Highest cycle time*	*Prob. Of lowest cycle time*	*Prob. Of highest cycle time*
All 3-vertex cases*	-	-	100%	-
All 4-vertex cases*	-	-	100%	-
Ten 5-vertex cases*	-	-	100%	-
5(5)	31	34	37.8%	62.2%
5(13)	23	25	54.7%	45.3%
Four 6-vertex cases*	-	-	100%	-
6(1)	31	35	42.8%	4.5%
6(3)	36	47	46.0%	2.0%
6(4)	50	62	75.8%	24.2%
6(5)	39	50	67.0%	0.2%
6(6)	46	64	50.2%	0.5%
6(7)	48	54	92.1%	7.7%
6(10)	39	41	32%	68%
Three 7-vertex cases	-	-	100%	-
7(1)	28	31	75.8%	0.1%
7(3)	36	41	99.2%	0.5%
7(4)	29	31	79.8%	0.5%
7(5)	39	41	72.6%	9.8%
7(8)	42	44	81.1%	18.9%
7(9)	52	58	44.1%	0.1%

*no solution is found for 2 of the 3-vertex cases, 2 of the 4-vertex cases, 1 of 5-vertex cases and 1 of 6-vertext cases.

As can be seen from Table 1, the basic algorithm has good effectiveness in computing a cyclic route which satisfies the relative deadlines of the vertices and produces short cycle times. For all 3-vertex cases, the algorithm generates the same cycle time in every run except in two graphs where no solution is found. For all 4-vertex cases, the algorithm also generates the same cycle time in every run except in two graphs where no solution is found. For 10 of the 5-vertex graphs, 4 of the 6-vertex cases and 3 for the 7-vertex cases, the algorithm generates the same cycle time in 1000 runs. For the other cases where a solution is found, the probability of producing the lowest cycle time based on the 1000 runs varies from 32% to 99.2%. Overall, the probability of getting a solution with the lowest cycle time based on the 1000 runs is generally high.

Out of the 57 tested graphs, our algorithm does not return a solution for 6 graphs. Checks carried out confirm that they are the cases where there is no solution for one UAV. This means one UAV is not able to make cyclic visits that satisfy the relative deadlines of all vertices. For example, the third instance of the 3-vertex graphs shown as case (iii) in Fig. 10 has no solution. In this graph, the relative deadlines of both vertices *b* and *c* are 10, and the weight of the edge *ac* is half of the relative deadline of vertex *c*. Travelling from vertex *c* to *a* would result in both relative deadlines of vertices *b* and *c* to be 5, which would only give enough time to satisfy either vertices *b* or *c* but not both. The outcome is the same even if travelling from vertex *c* to *a* is avoided. Hence, it is impossible for a single UAV to satisfy both constraints. Therefore, even though when the algorithm fails to find a solution, there is no guarantee that one does not exist, our test results show that the basic algorithm is quite competent.

TABLE II. CPU TIME (MILLISECONDS) OF BASIC ALGORITHM

Category	*Min*	*Median*	*Max*	*Mean*
3-vertex cases that have a solution	0.0209 - 0.0431	0.0233 – 0.0478	0.1043 – 0.6068	0.0414
4-vertex cases that have a solution	0.0332 – 0.1150	0.0387 – 0.2839	0.1059 – 3.0534	0.1044
5-vertex cases that have a solution	0.0474 – 0.1861	0.0577 – 0.5914	0.1798 – 5.8114	0.2101
6-vertex cases that have a solution	0.0628 – 0.2821	0.0790 – 0.9256	0.2441 – 13.2342	0.3826
All 7-vertex cases (have a solution)	0.1075 – 0.4618	0.1316 – 28.0597	0.4930 – 327.2841	5.0229
3-vertex cases that do not have a solution	34.5742 – 34.7666	36.0654 – 36.9703	50.1992 – 260.7659	44.4364
4-vertex cases that do not have a solution	73.0449 – 95.0147	74.6350 – 99.2205	89.9638 – 144.5263	88.8542
5-vertex case that does not have a solution	111.6120	115.6246	138.9673	117.5761
6-vertex case that does not have a solution	97.4747	101.7835	352.0034	110.8454

Table 2 shows the CPU time taken by the basic algorithm in the 1000 independent runs for each tested graph. Note that in the basic algorithm, if one trial does not find a solution, it will try another time until either a solution is found or the algorithm has tried 1000 times. We present the CPU time based on the number of vertices in the graphs and whether a solution is found for a graph. The *Min/Median/Max* columns show the minimum/median/maximum CPU time in the independent runs for each category of graphs. The *Mean* column gives the overall average CPU time of all runs for graphs in the category. The results in Table 2 show that the basic algorithm is very fast and efficient. Most of the cases will take not more than a few milliseconds, sometimes a few tens of milliseconds to find a solution. The longest time on one case takes 372 milliseconds to compute a solution. When a solution cannot be found, the algorithm will go through 1000 trials before reporting failure. In these cases, the computational time ranges from a few tens to a few hundreds of milliseconds.

Our improved algorithm is able to produce solutions of better quality. Table 3 shows the cycle time of the cyclic routes computed by the improved algorithm. It does not include cases where all independent runs produce the same cycle time since the improved algorithm returns the same solution as the basic

algorithm for these cases. Compared with Table 1, Table 3 shows that the highest cycle time obtained from the 1000 independent runs is smaller than those obtained by the basic algorithm. For example, case 6(3), in Table 1 the highest cycle time is 47 while in Table 3, it is 42. This means the gap between the lowest cycle time and the highest cycle time obtained by the algorithm is smaller. In the case of 6(3), it means when the improved algorithm is executed, the cyclic route found will have a cycle time in the range from 36 to 42 instead of the range from 36 to 47. The probability of getting the lowest cycle time based on 1000 independent runs is significantly improved. In the case of 6(3), the probability of getting a cyclic route of the lowest cycle time, 36, is 46% when the basic algorithm is run. When the improved algorithm is run, the probability is 91.9%. Compared with the corresponding results in Table 1, we see significant improvements.

TABLE III. CYCLE TIMES OF SOLUTIONS FROM IMPROVED ALGORITHM

case	***Lowest cycle time***	***Highest cycle time***	***Prob. Of lowest cycle time***	***Prob. Of highest cycle time***
5(5)	31	34	75.8%	24.2%
5(13)	23	25	86.1%	13.9%
6(1)	31	33	94.6%	0.6%
6(3)	36	42	91.9%	5.9%
6(4)	50	62	98.8%	1.2%
6(5)	39	41	97.1%	2.9%
6(6)	46	48	86.8%	13.2%
6(7)	48	48	100%	0%
6(10)	39	41	67.1%	32.9%
7(1)	28	29	98.7%	1.3%
7(3)	36	36	100%	0%
7(4)	29	30	99.1%	0.9%
7(5)	39	41	98.7%	0.2%
7(8)	42	44	99.6%	0.4%
7(9)	52	54	85.8%	0.4%

Table 4 shows the CPU time taken by the improved algorithm in the same format as Table 2. Only the cases where a solution can be found are shown since it takes the same amount of time for the improved algorithm to realize no solution can be found as for the basic algorithm. Naturally a longer computational time is required by the improved algorithm. However, all cases need less than half a second.

TABLE IV. CPU TIME (MILLISECONDS) OF IMPROVED ALGORITHM

Category	***Min***	***Median***	***Max***	***Mean***
3-vertex cases that have a solution	0.0648 - 0.1335	0.0676 – 0.1454	0.2054 – 0.5353	0.1170
4-vertex cases that have a solution	0.1035 – 0.3560	0.1165 – 0.9092	0.2679 – 3.9392	0.3120
5-vertex cases that have a solution	0.1509 – 0.8593	0.1758 – 2.4899	0.3642 – 15.7958	0.8812
6-vertex cases that have a solution	0.2141 – 1.0327	0.5199 – 3.4989	1.7031 – 20.7731	1.6643
7-vertex cases that have a solution	0.2781 – 8.6815	0.4229 – 126.2340	1.0003 – 402.2829	14.8122

V. CONCLUSIONS

The cyclic routing problem for a single UAV is to find a cyclic route for the UAV to visit all target areas possibly multiple times such that all relative deadlines are observed. The solution to such a problem may be exponential in length. We propose a randomized heuristic algorithm to computes a solution that has a reasonably short cycle time for a given graph with given relative deadlines. Even though we cannot guarantee to find a solution when there is one, the effectiveness of the algorithm is shown by the good quality solutions for 51 graphs out of 57 graphs. The remaining 6 graphs are the ones where there is no solution for a single UAV. The efficiency of the algorithm is good. The CPU time required by the algorithm is in milliseconds for all our tested cases.

Further tests can be conducted with cases of higher number of target areas.

REFERENCES

[1] N. Basilico, N. Gatti and F. Amigoni, "Developing a deterministic patrolling strategy for security agents," In Proceedings of the 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Vol. 02, pp. 565-572. IEEE Computer Society, 2009.

[2] N. Basilico, N. Gatti and F. Amigoni, "Patrolling security games: Definition and algorithms for solving large instances with single patroller and single intruder," Artificial Intelligence, Vol. 184–185, pp. 78–123, June 2012.

[3] P. J. Bristeau, F. Callou, D. Vissière and N. Petit, "The Navigation and Control technology inside the AR.Drone micro UAV," In Proceedings of the 18th IFAC World Congress, pp. 1477-1484, August 2011.

[4] N, Drucker, M. Penn and O. Strichman, Cyclic Routing of Unmanned Aerial Vehicles, Lecture Notes in Computer Science: Integration of AI and OR Techniques in Constraint Programming, Vol. 9676, pp. 125-141, 2016.

[5] G. H. Elkaim, F. A. Lie, and D. Gebre-Egziabher, "Principles of Guidance, Navigation and Control of UAVs," In Handbook of Unmanned Aerial Vehicles, K. P. Valavanis, & G. J. Vachtsevanos (Eds.), Springer, 2015, pp. 347-380.

[6] J. J. Enright, E. Frazzoli, M. Pavone and K. Savla, "UAV Routing and Coordination in Stochastic, Dynamic Environments," In Handbook of Unmanned Aerial Vehicles, K. P. Valavanis, & G. J. Vachtsevanos (Eds.), Springer, 2015, pp. 2079-2109.

[7] J. L. Fargeas, B. Hyun, P. Kabamba and A. Girard, "Persistent Visitation under Revisit Constraints," In Proceedings of International Conference on Unmanned Aircraft Systems, pp. 952-957, 2013.

[8] H. Ho and J.Ouaknine, "The cyclic-routing UAV problem is PSPACE complete," Lecture Notes in Computer Science: Foundations of Software Science and Computation Stuctures, Vol. 9034, pp. 328-342. Springer, 2015.

[9] F. Kendoul, "Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems," J. Field Robotics, Vol. 29, pp. 315–378, 2012.

[10] V. Mersheeva and G. Friedrich, "Multi-UAV Monitoring with Priorities and Limited Energy Resources," In Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, pp. 347-355, 2015.

[11] C. Park, Y. Kim, B. Jeong, "Heuristics for determining a patrol path of an unmanned combat vehicle," Computers & Industrial Engineering, Vol. 63, pp. 150–160, 2012.

[12] A. J. Pohl and G. B. Lamont, "Multi-Objective UAV Mission Planning Using Evolutionary Computation," In Proceedings of the 2008 Winter Simulation Conference, pp. 1268-1279, 2008.