Additional Appendix

Dimitris Chaikalis[1*]

[1*]Electrical & Computer Engineering, New York University, Brooklyn, 11201, New York, USA.

Corresponding author(s). E-mail(s): dimitris.chaikalis@nyu.edu;

1 Algorithms for Structure Analysis

1.1 Edge-List Generation Algorithm

The algorithm for generating the weighted edge-list $\mathcal{E}$ and the look-up table of individual rotations, can be found in the pseudocode segment below.

While explaining the entire code in depth would not be practical, the idea is to start from a polygon and then explore the entire structure from there. This is why a new recursion is initiated every time a new polygon is encountered, such that each call to the "Process()" function will in the end process all edges of its respective polygon.

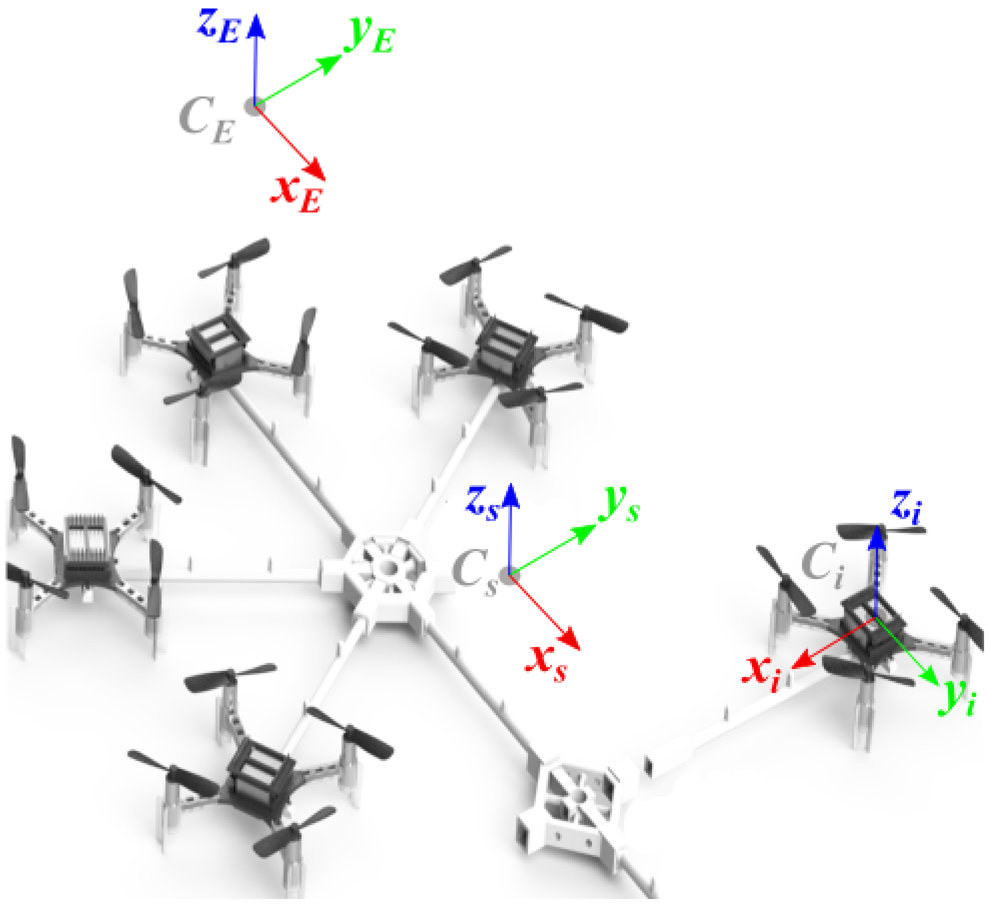
The user only needs to describe the entire structure as a simple sequence of characters, with letters signifying a polygon, numbers signifying the respective face of said polygon, a closing bracket signifies that the previous polygon is the starting one and a closing parenthesis signifies that all faces of the current polygon have been accounted for.

So for example, if the graph corresponding to the structure of Figure 1.1 is required, the user simply enters the word **"h]1s314)2345)"** and the program will decode this word and arrive at the node-list, edge-list and rotation look-up table for the entire graph.

This means that the program will begin from the hexagon, understand that this is the starting point, note that face **1** is utilised and that a link connects this face of the hexagon to a square. Then, a recursion is initiated, where the program notes that the previous link arrives at the square through face **3**, constituting an edge between hexagon and square, as well as faces **1, 4** being utilised with no further polygons, followed by a closing parenthesis, signifying

Algorithm 1 Process

```
procedure PROCESS(word, start, link_length)
    polygons=0, L←[], INIT←[], d=0, First=true, $\Phi = 0$, count=0
    n_st = start + string(polygons)
    polygons++
    Node_List.Append(n_st)
    for c in word do
        count++
        if d>0 then
            d− = 1
            continue
        end if
        if First then
            if c!=']' then
                INIT = c
            end if
            First = false
        else
            if c.isNumber() or c==')'  then
                if !L.isEmpty() then
                    n_L = string(L) + '_' + n_st
                    Node_List.Append(n_L)
                    dist = link_length
                    Edge_List.Append([n_st,n_L,dist]
                    Rotations_Table.Append([n_st, n_L, $\Phi$]
                end if
                if c.isNumber() then
                    L = c, $\Phi$ = fixed_rotation(start, INIT, L)
                else
                    break
                end if
            else if c.isLetter() then
                n_C = c + '_' + string(polygons)
                d = Process(word[count:], c)
                Edge_List.Append([n_st, n_C, link_length]
                Rotations_Table.Append([n_st, n_C, $\Phi$])
                L←[]
            end if
        end if
    end for
    return count
end procedure
```



that nodes are also present at the ends of the links placed at faces **1, 4** of the square.

The second recursion exits, returning the number of characters it had to go through, so that the current recursion knows how much of the word it needs to skip, until it is back at the hexagon. Then numbers **2. . . 5** followed by a closing parenthesis, indicate that there are nodes present at the ends of these links as well. Finally, the algorithm terminates.

The Node_List, Edge_List and Rotations_Table can then be handed over to the main algorithm of Section III, to compute all displacements and rotations of the structure agents, with respect to the hexagon.

1.2 Graph Traversal Recursion Algorithm

Clarifications: the main function only requires the number of nodes and the edge list, which is composed of elements of the type (f_node, s_node,

Algorithm 2 Check_Node

```
procedure Check_Node(node_num, Except_list, d, )

    counter = 0
    Except_list.Append(node_num)
    for $i$ = 1 to Length(Adj) do
        $w$ = Adj[node_num][$i$]
        if $w > 0$ and $i \notin$ Except_list then
            counter++
            $\Phi = \Phi^{-}+$ get_rot(node_num,i)
            d = $d^{-} + R_{z,\Phi}[-w, 0, 0]^T$
            Check_Node($i$, Except_list, $\Phi$, d, Adj)
        end if
    end for
    if counter == 0 comment: is leaf node then
        num_of_agents++
        Agents_list.Append(Agent(num_of_agents, $\Phi^{-}$, d))
    end if
    return
end procedure
```

Algorithm 3 Main

```
procedure Main

    Adj = get_adjacency($\mathcal{E}$, num_of_nodes)
    Check_Node(0, [], 0, [0,0,0], Adj)
end procedure
```

link_length). Then helper functions get_adjacency and get_rot are used to create the adjacency matrix, with link_length as weight of edge, and utilise a look-up table to search for the rotation between successive nodes (easy to create due to usage of fixed polygons) respectively. Finally, once all recursions are finished, every leaf node (aka drone) has added itself with its rotation and displacement, to the Agents_list.