Algorithm: Convex Partitioning and Mapping algorithm

Input :

1. **A directed application graph G(V, E)**
2. **A directed resource graph R(V', E')**
3. **Value of seed // for initial random k-way partitioning.**

Output: A local optimum partition set $\pi = (P_1, \bullet\bullet\bullet, P_k)$ with minimum cutsize of π and minimum longest depth L_{nc} of non-convex partition set $p_{nc} \in \pi$.

```
k ← n/n' + 1;   // n = |V| and n' =|V'|
Compute d_max and G_max of graph G(V, E).
bucketsize ← 2* G_max +1;
itr ← 0;
do
   Create and set a random seed for randomize partitioning.
   Produce an initial random k-way partition set π = (P_1, • • •, P_k).
   Cutsize ← currentCutsize;
   do
       Compute node gain for each node and initialize all nodes as unlocked.
       Build bucket array based on move gain of nodes.
       nlock ← 0;   // the number of locked node=0.
       do
           Choose a legal node l ∈ V with maximum move gain.
           Delete node l from bucket array and lock it.
           Make the node move tentatively.
           Update the cost and move gain arrays of all affected nodes.
           nlock ← nlock+1;
       while (nlock < no_nodes);
       Find the maximum gain sum S_b for selected b nodes.
       if (S_b > 0) then
```

Make the first b nodes moves permanent.

Cutsize ← Cutsize - S_b;

end if

while(Sb> 0);

Compute longest depth L_{nc} and L_c of non-convex partition set $P_{nc} \in \pi$ and convex partition set $P_c \in \pi$ respectively.

itr ← itr+1;

while ($L_{nc} > L_c$ and itr < MAXIMUM_ITERATION);

===

===

Algorithm: Initial_Partition Algorithm

Input: G(V, E), R(V', E') and k

Output: An initial k-way partition set $\pi = (P_1, \bullet\bullet\bullet, P_k)$

for i←0 to k do

P_i.current_size←0;

end for

for i←0 to |V| do

min_size← P_0.current_size; //assign the current size of 0th partition to min_size.

for j←0 to k do

if (min_size> P_j.current_size) then

min_size← P_j.current_size;

min_inx←j; // min_inx indicates the partition whose size is minimum.

end if

end for

for j←0 to k do

if (P_j.current_size= min_size) then

tcount←tcount+1; //count the number of partitions whose size is equals to min_size.

end if

end for

min_inx ←Random (0, tcount);

Assign node v_i to partition P_{min_inx}.

P_{min_inx}.current_size←P_{min_inx}.current_size+1;

end for

for i←0 to k do

P_i.max_size←|V'|;

P_i.minimum_size←0;

end for

==

==

Algorithm: Longest_Depth Algorithm

Input: A local optimum partition set π = (P_1, • • •, P_k) and G(V, E)

Output: **Longest depth L_{nc} of non-convex partition set, longest depth L_c , minimum longest depth L_{mc} of convex partition set and number of non-convex partitions N_{ncp}**

L_c←0;

L_{nc}←0;

L_{mc}←∞;

N_{ncp}←0;

for i←0 to k do

flag←0;

L_i←0; // L_i is longest depth of partition P_i

for each node v_{ip}∈P_i do

if (v_{ip} is input node of P_i) then

for each node v_{op}∈P_i do

if (v_{op} is output node of P_i and $v_{ip} \neq v_{op}$) then

local_longest_depth←0;

```
                part_no← i;
                Init_Dfs(v_ip, v_op, part_no, G, local_longest_depth, flag);
                if(local_longest_path> L_i) then
                  L_i←local_longest_path;
                end if
              end if
            end for
          end if
        end for
        if (flag=1 and L_i > L_nc) then
          L_nc ← L_i;
        end if
        if (flag=0 and L_i > L_c) then
          L_c ← L_i;
        end if
        if (flag=0 and L_mc > L_i) then
          L_mc ← L_i;
        end if
        if (flag=1) then
          N_ncp ← N_ncp +1;
        end if
      end for
```

==

==

Algorithm: Init_Dfs Algorithm

Input: Input node v_{ip}**, output node** v_{op}**, part_no and G(V, E)**

Output: Returns Longest depth between input, output node and flag value

```
    Create an array visited of size |V|.    //to check whether a node is visited or not.
```

```
Create an array path of size |V|.     // to store all nodes of a path.
path_index←0;      //it  indicates the current node in the path array.
for i←0 to |V| do
   visited[i] ← False;
end for
Dfs(v_ip, v_op, part_no, G, visited, path, path_index, local_longest_depth, flag);
```

==

Algorithm: Dfs Algorithm

Input: Current node u, output node v_{op}, part_no and G(V, E), visited[], path[] and path_index

Output: Longest depth between input, output node and flag value

```
visited[u] ←  True;  //mark the current node as visited.
path[path_index] ←  u;  //add u to path
path_index←path_index+1;
if (u= v_op) then    // If current vertex is same as output node.
   for i←0 to path_index do
      if (path[i] ∉ P_part_no) then
         flag←1;
         exit for
      end if
   end for
   if ((path_index-1)> local_longest_path) then
      local_longest_path←(path_index-1);
   end if
else              // If current node is not destination.
   for each nodes w adjacent to current node u do
      if (visited[w]=False) then
         Dfs(w, v_op, part_no, G, visited, path, path_index, local_longest_path, flag);
      end if
```

```
    end for
end if
Remove current node u from path[] and mark it as unvisited;
```