Khuyen Le Thi Minh s5128 – ASD 6

TASK DESCRIPTION:

The clique in a graph G=(V,E) is a maximal set of graph's vertexes in which any two vertex are connected via an edge form the set E. Design a greedy algorithm that finds a clique in an input graph. Give a detailed description of your method and estimate its complexity.

ALGORITHM:

In order to find all cliques in a graph, we will first find all the possible subsets from the graph's vertexes, after that we'll check if the subset is a clique, and after that we'll eliminate the cliques that are the part of other clique in order to find collection of maximal cliques.
Since the power set of n vertexes graph would be 2^n, with n as a bigger number, there would be a lot of calculation so we will first find the collection of vertexes that is connect to the given vertex in a graph and after that find the power set of that collection, in this situation only in the worst case, when the graph is a complete one, the calculation for power set would be 2^n.

```
//we will construct the structure Edge that contain two nodes: begin and end
struct Edge{
   node begin;
   node end;
};
//we will construct the structure of graph that contain two arrays: the one with all vertexes and other one with all
edges
struct Graph{
   node[] points;
   Edge[] edges;
};

//the function that will generate all power sets for the given input linked list a and will return the list of list. Since the
power set of n element would be all the subsets of n-1 elements (without the first element) and subsets of n-1
elements that additionally contain the first element.
//Since the function will generate 2^n element for an n-elements input list the time complexity for this function will
be O(2^n)
pSet(List a){
//if the lenght of the input list is 0 then we'll add the empty list to the result (in the next code with that empty list
we'll generate the list of single element set which are the first element of each list that go through the function), this
is a base condition for the function to stops, since once we emptied the all elements in a we will go back.
        if (lenght of a == 0) then
        {
                result add empty list
        }fi;
        else then
        {
//first we'll take the first element of the list
                first = a[0];
//after that with each element (a list) in the result of the list without first element that go through the recursive
function we add the first element in
                for (s in pSet(s[from a[1] to lenght(a)])) do
                {
                        withFirst = [s];
                        withFirst add first;
//after that to the result list we add the subsets that contain the first element and those that do not contain the first
element.
                        result add withFirst;
                        result add s;
                }od;
        }
        return result;
};

//We now will have the function to find all maximal cliques in the input graph a that is not an empty with at least 2
nodes and those 2 nodes are connect by an edge.
Clique(Graph a){
//We will have a list that each element in that list will be two elements input, the first one is the node from the
vertexes of the graph, the second one will be the list of all nodes that are connect to that node by edges
        tempResult;
```

```
//We will have a list of list that contains all possible subsets of the sets in tempResult, some of them are cliques, but some not
        possibleCombinations;
//We will have a list of list that contains all possible maximal cliques
        possibleCliques;
//We will have a counter that would hold the number of nodes that are connected to all other nodes in the graph, this will help us to check if the graph is complete one so we would avoid the worst case in time complexity. The counter will be set on default as 0.
        int counter0 = 0;
//now we will take each of the node from the graph and find all possible nodes that are connected to it, this loop will run through n elements so the time complexity of its would be O(n)
        for(i in vertexes of a) do
        {
//We create an empty List of nodes that will contains all nodes connected to i
                temp;
//Now we will run through all edges in the graph to find the nodes that are connected to i, this loop will run through max(n*(n-1)/2)-for complete graph times so its time complexity would be O(n*(n-1)/2)=O(n^2). The time complexity for this double loops would be O(n*(n*(n-1)/2)) = O(n^3)
                for(j in edges of a) do
                {
//We check if the j has one of its node that is equals to i, if yes then we add the second node of j to temp
                        if(i = node begin of j) then
                        {
                                temp add end of j;
                        }
                        else if (i = node end of j) then
                        {
                                temp add begin of j;
                        }fi;
                }od;
//We check if temp is equals to number of nodes in a - 1, if yes we'll increase our counter0 by 1
                if(temp == number of vertexesin a - 1) then
                {
                        counter0++;
                }
//We put the node i and the list of nodes that are connected to it into tempResult
                tempResult add (i, temp);
        }od;
//Now we'll check if our counter0 is equals to number of nodes in the input graph, if yes then we have a complete graph so we'll return the nodes in graph a as the result, if the graph is not a complete one we'll generate the power sets for its nodes. This will prevent the worst case of time complexity
        if(counter0 == number of vertexes in a) then
        {
//Since we would return as the result the list of list possibleCliques, we'll create the empty list temp and copy all nodes in a to it, after that we'll add it to possibleClique. The time complexity for this loop would be O(n)
                for(s in vertexesof a)
                {
                        temp add s;
                }
                possibleClique add temp;
        }
        else then
        {
//now we will go through each element of the tempResult, we will take the node (i) and list of nodes (temp) that are to it connected, first we'll generate the power set for temp and after that for each set in those subsets we will add the i in. This outer loop will run n times - O(n), since tempResult should be n-elements list
                for(m in tempResult) do
                {
//to generate the power set as we state before, it would cost maximal 2^n in time complexity, so the inner loop would run maximum 2^n times - O(2^n), so the time complexity for this double loops would be O(n*2^n)
                        for(s in pSet(list temp of m)) do
                        {
                                s add i of m;
//now we will add each new set that possible is a clique to the possibleCombination
                                possibleCombinations add s;
                        }
```

```
                }od;
//now we will run through the possibleCombinations and will check for each subset if it's a clique, after that will check if it's a maximal clique. Since possibleCombinations can have elements with upper bound n*2^n so the worst case time complexity for this outer loop would be O(n*2^n). The time complexity for this loop with its inner loops would be O(n*2^n * (n^4 + n^2*3^(n/3))) = O (n^5*2^n + n^3*24^(n/3)) = O(n^3*24^(n/3))
                for (m in possibleCombinations) do
//we will have the counter that will set at beginning to zero, this counter will hold the number of time each node in the combination is connects to other nodes in the same combination. If in the end the counter would be equals to number of edges that the combination should have if it's a complete subgraph then it'd be a clique.
                        int counter1 = 0;
//Now we go through the nodes in the combination until the one before last. The worst case time complexity for this loop would be O(n)
                        for(i from 0 to size of m -1) do
                        {
//now we go again through the nodes in the combination start from i+1- to compare to nodes from next to the one we pick. The worst case time complexity for this loop would be O(n)
                                for(j = i+1 to size of m) do
                                {
//now we check if i and j are connect by edge in input graph if yes then we increase counter1 by 1. The worst case time complexity for this loop would be O(n*(n-1)/2) = O(n^2) so the worst case time complexity for those 3 loops would be O(n*n*n^2) = O(n^4)
                                for(h in edges of a) do
                                {
                                        if((i == begin node of h && j == end node of h) ||
                                        (i == end node of h && j == begin node of h)) then
                                        {
                                                counter1++;
                                        }fi;
                                }od;
                        }od;
                }od;
//now we check if the counter would be equals to number of edges if the combination is an complete subgraph
                if(counter1 == (size of m * (size of m -1) /2)) then
                {
//at the beginning we'll add the first clique to the possibleCliques
                        if(size of possibleCliques == 0) then
                        {
                                possibleCliques add m;
                        }
//after that we'll check if the next cliques that we will add are the maximal
                        else then
                        {
//we generate the boolean value to check if the clique is a duplicate one and we set it by default as true
                                boolean isD = true;
//now we'll go through the possibleCliques to check if our m that is a new clique that we found and want to add in the result is a duplicate one. The time complexity for this loop would be O(3^(n/3)) since in worst case there would be 3^(n/3) (according to a result of Moon & Moser)
                                for(i in possibleCliques) do
                                {
//we'll create a boolean value to check if for each combination in possibleCliques m is duplicate one or not and we set it by default to true.
                                        boolean isDupliacte = true;
//we generate another value counter to hold the number of nodes that are the same in m and i
                                        counter2 = 0;
//for each node in m. The time complexity for this loop in worst case would be O(n)
                                        for (l in m) do
                                        {
//for each node in i. The time complexity for this loop in worst case would be O(n). The time complexity in those 3 inner loops would be O(n^2* 3^n)
                                                for(k in i) do
                                                {
//we check if they're the same, if yes then we'll increase our counter2 by 1, since the nodes in combinations are unique so once we find the same node we would break out of the inner loop to go to next clique in possibleCliques.
                                                        counter2++;
                                                        break;
                                                }od;
```

```
                                        }od;
//now we compare the length of the combination i and m as well as the counter2. In the case that the length of m, i
and counter2 are the same or the length of i is equals to counter2 and is smaller than the length of m (so that m would
be maximal clique instead of i), we replace i by m and set isDuplicate to true.
                                        if((length of i < length of m && counter2 == length of i) ||
                                        (length of i == length of m && counter2 == length of i)) then
                                        {
                                                replace i with m;
                                                isDuplicate = true;
                                        }
//in the case that the repeated number of nodes in i and m that is counter2 is smaller than the smaller length of those
two set we set isDuplicate to false
                                        else if((length of i < length of m && counter2 < length of i) ||
                                        (length of i > length of m && counter2 < length of m) ||
                                        (length of i == length of m && counter2 < length of i)) then
                                        {
                                                isDuplicate = false;
                                        }
//in any other case we set isDuplicate to true
                                        else then
                                        {
                                                isDuplicate = true;
                                        }fi;
//if we find the same set as m in the possibleCliques we set the isD to true and break the loop
                                        if(isDuplicate) then
                                        {
                                                isD = true;
                                                break;
                                        }fi;
                                }od;
//if the set m is not in possibleCliques we will add it in
                                if(!isD) then
                                {
                                        possibleCliques add m;
                                }fi;
                        }fi;
                }fi;
        }od;
//we'll return the result list.
        return possibleCliques;
};

//We will create the algorithm to find all the maximum cliques in the input graph
maximumClique(Graph a)
{
// first we'll create the list of list that will contains all maximal cliques in the input graph
  allCliques;
//we'll now create the result list of list that will conatains all possile maximum cliques in the graph
  maximumCliques;
//we will have an integer that will hold the size of the maximum cliques
  int max = 0;
//now we'll go through all the cliques and find the size of maximum cliques. The time complexity for this loop would
be O(3^(n/3)) since in worst case there would be 3^(n/3 )(according to a result of Moon & Moser)
  for (i in allCliques) do
  {
//we check the size of i, if it's bigger than max then we replace the number in max with size of i
    if(size of i > max) then
    {
      max = size of i;
    }fi;
  }od;
//now we'll go through all the cliques and find the maximum cliques with the size that equals to max then add it to
result list maximumCliques. The time complexity for this loop would be O(3^(n/3)) since in worst case there would
be 3^(n/3 )(according to a result of Moon & Moser)
for (i in allCliques) do
  {
```

```
      if(size of i == max) then
      {
         maximumCliques add i;
      }fi;
   }od;
//we'll return the result list.
   return maximumCliques;
}
```

Justification of an algorithm's correctness:

The specification of the algorithm pSet becomes a pair <WP, WK>, where WP = (The list that contains finite number of nodes) and WK = (All subsets created from the list of nodes).

The specification of the algorithm Clique becomes a pair <WP, WK>, where WP = (The graph that has at least 1 node and the number of nodes and edges of the graph are the finite numbers) and WK = (The list of all possible maximal clique in the input graph).

The specification of the algorithm maximumClique becomes a pair <WP, WK>, where WP = (The graph that has at least 1 node and the number of nodes and edges of the graph are the finite numbers) and WK = (The list of all possible maximum clique in the input graph).

Stop condition of the algorithm – for pSet algorithm we have base condition to stop is when the input size of the list is equals 0 (since the input list has a finite number of nodes, it will reach the 0). The function will be call recursively until all the first element of the list are removed, then it will generate the empty list and will go back.

For Clique algorithm with the first loop it will go through all vertexes in iput graph, since the number of vertexes is a finite one the loop will end. The result of the first loop would be a finite number. Since each loop that will go through the number of vertexes or edges in input graph or the finite list result that had been created by other loosp or pSet algorithm is a finite number of list then the algorithm will stops.

For maximumClique algorithm, since the result of the Clique algorithm will be a finite list and the algorithm contains two loops going through that list, the algorithm will stops.

Partial correctness of the algorithm – for pSet with initial condition that there input wild be a list of nodes, in the case there is an empty list the result list of lists would be also an list with a empty list. In the case there is at least 1 element in the list the algorithm will remove the first element of the list and call recursive function until all the element in the list will be remove then it will create an empty list and add it to result list, when the algorithm going back it each time it will duplicate the lists already in result list and add in those list the first element that we removed. With this we'll get in result all the subsets of the input list of nodes in the case the list is not an empty one or an empty list if the input list is empty. For example with the list of elements (1, 2, 3) the algorithm will remove the first element of the list (1) and put it as 'first', then will call the recursive function for new created list(2,3), the algorithm will remove the element (2) and put it as another 'first', then will call recursive function for new list (3), this time the element (3) will be remove and put it as the new 'first' and the recursive function will be call for an empty list. We now will create an empty list and add it to result list of lists and go back. This time we'll copy the empty list in result list and add (3) in, so in result list will be :{}, {3}, after that we'll go back again and copy all the sub-lists in result list and add (2) to it so now in result list will be: {}, {3}, {2}, {3,2}, after that we'll go back again and duplicate all the sub-lists in result list and add (1) to it so now the algorithm will end and in result list we'll have: {}, {3}, {2}, {3,2},{1}, {3,1}, {2,1}, {3,2,1} which are all the subsets of the set {1,2,3}

For Clique algorithm for the initial condition WP = (The graph that has at least 1 node and the number of nodes and edges of the graph are the finite numbers) and WK = (The list of all possible maximal clique in the input graph).
In the first loop we'll generate the list of all nodes in graph and the nodes that are connected to each node.
We then will check if the graph is an complete graph, if yes we'll return the list pf nodes of the graph as the result if not then we'll generate the subsets for each node and the list of nodes that is connected to its. Those subsets are all possible subgraph in the input graph.
We then will check if the subgraph is the complete graph. If the subgraph is the complete one then is a candidate to become a maximal clique.
After that we will try to check for the duplicated subgraph and remove all the cliques that are not a maximal one.
In the result we will get the list of maximal cliques in the input graph.
Because the Clique algorithm is partially correct in terms of specification <WP,WK> and we have shown the stop condition for this algorithm, it is also absolutely correct with respect to the considered specification.

For maximumClique algorithm for the initial condition WP = (The graph that has at least 1 node and the number of nodes and edges of the graph are the finite numbers) and WK = (The list of all possible maximum clique in the input graph).
First we'll generate all possible maximal clique in the graph, then we'll go through it to find the size of the maximum cliques (in the case there are 2 or more maximum clique) then we'll store that number in variable max.
After finding max, we'll go through all the maximal cliques again to check the cliques that the size is the same as max and add it to the result list.
Because the maximumClique algorithm is partially correct in terms of specification <WP,WK> and we have shown the stop condition for this algorithm, it is also absolutely correct with respect to the considered specification.

Implementation of algorithm in Java:

```
class Edge {
   public char begin;
   public char end;
   public Edge(char begin, char end){
      this.begin = begin;
      this.end = end;
   }
   public char getBegin(){
      return begin;
   }
   public char getEnd(){
      return end;
   }
}
class Graph {
   public char[] points;
   public Edge[] edges;
   public Graph(char[] points, Edge[] edges){
      this.points = points;
      this.edges = edges;
   }
   public char[] getPoints(){
      return points;
   }
   public Edge[] getEdges(){
      return edges;
   }
}
public class ASD6{

   public static List<List<Character>> pSet (List<Character> a){
      List<List<Character>> result = new ArrayList<>();
      if(a.size() == 0){
         result.add(new ArrayList<>());
      }
      else{
         char first = a.get(0);
         for(List<Character> s : pSet(a.subList(1, a.size()))){
            List<Character> withFirst = new ArrayList<>(s);
            withFirst.add(first);
            result.add(withFirst);
            result.add(s);
         }
      }
      return result;
   }
   public static List<List<Character>> Clique(Graph a){
      HashMap<Character, List<Character>> tempResult = new HashMap<>();
      List<List<Character>> possibleCombinations = new ArrayList<>();
      List<List<Character>> possibleCliques = new ArrayList<>();
      int counter0 = 0;
      for(int i = 0; i < a.getPoints().length; i++){
```

```
    ArrayList<Character> temp = new ArrayList<>();
    for(int j = 0; j < a.getEdges().length; j++){
      if(a.getPoints()[i] == a.getEdges()[j].getBegin()){
        temp.add(a.getEdges()[j].getEnd());
      }
      else if(a.getPoints()[i] == a.getEdges()[j].getEnd()){
        temp.add(a.getEdges()[j].getBegin());
      }
    }
    if(temp.size() == a.getPoints().length-1){
      counter0++;
    }
    tempResult.put(a.getPoints()[i], temp);
  }
  if(counter0 == a.getPoints().length)
  {
    List<Character> temp = new ArrayList<>();
    for(char i : a.getPoints())
    {
      temp.add(i);
    }
    possibleCliques.add(temp);
  }
  else
  {
    Set set = tempResult.entrySet();
    Iterator iter = set.iterator();
    while(iter.hasNext()){
      Map.Entry<Character, List<Character>> me = (Map.Entry)iter.next();
      for(List<Character> s : ASD6.pSet(me.getValue())){
        s.add(me.getKey());
        possibleCombinations.add(s);
      }
    }
    for(List<Character> m : possibleCombinations){
      int counter = 0;
      for(int i = 0;i < m.size()-1; i++){
        for(int j = i+1; j < m.size(); j++){
          for(int h = 0; h < a.getEdges().length; h++){
            if(m.get(i) == a.getEdges()[h].getBegin() && m.get(j) == a.getEdges()[h].getEnd() ||
                m.get(j) == a.getEdges()[h].getBegin() && m.get(i) == a.getEdges()[h].getEnd()){
              counter++;
            }
          }
        }
      }
      if(counter == ((m.size()*(m.size()-1))/2))
      {
        if(possibleCliques.size() == 0){
          possibleCliques.add(m);
        }
        else{
          boolean isD = false;
          for (List<Character> i : possibleCliques){
            boolean isDuplicate = true;
            int counter2 = 0;
            for(char l : m){
              for(char k : i){
                if(l == k){
                  counter2++;
                  break;
                }
              }
            }
            if((i.size() < m.size() && counter2 == i.size()) ||
                (i.size() == m.size() && counter2 == i.size())){
```

```
                    possibleCliques.set(possibleCliques.indexOf(i), m);
                    isDuplicate = true;
                  }
                  else if((i.size() < m.size() && counter2 < i.size()) ||
                       (i.size() > m.size() && counter2 < m.size()) ||
                       (i.size() == m.size() && counter2 < m.size())){
                    isDuplicate = false;
                  }
                  else{
                    isDuplicate = true;
                  }
                  if(isDuplicate){
                    isD = true;
                    break;
                  }
                }
                if(!isD){
                  possibleCliques.add(m);
                }
              }
            }
          }
        }
        return possibleCliques;
    }


    public static List<List<Character>> maximumClique(Graph a){
        List<List<Character>> allCliques = ASD6.clique(a);
        List<List<Character>> maximumCliques = new ArrayList<>();
        int max = 0;
        for(List<Character> i : allCliques)
        {
            if (i.size() > max)
            {
                max = i.size();
            }
        }
        for(List<Character> i : allCliques)
        {
            if (i.size() == max)
            {
                maximumCliques.add(i);
            }
        }

        return maximumCliques;
    }
}
```

Example 1:
The Graph with the array of nodes = {'A'} and empty array of edges.
The result for algorithm Clique will be 1 element list: {A}
The result for algorithm maximumClique will be 1 element list: {A}

Example 2:
The Graph with the array of nodes = {'A', 'B', 'C', 'D', 'E', 'F', 'G'} and an array of edges: {('A', 'B'), ('A', 'C'),('A', 'D'), ('A', 'E'), ('A', 'F'), ('A', 'G'), ('B', 'C'), ('B', 'D'), ('B', 'E'), ('B', 'F'), ('B', 'G'), ('C', 'D'), ('C', 'E'), ('C', 'F'), ('C', 'G'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'F'), ('E', 'G'), ('F', 'G')}
The result for algorithm Clique will be 1 element list: {ABCDEFG}
The result for algorithm maximumClique will be 1 element list: {ABCDEFG}

Example 3:

The Graph with the array of nodes = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'} and an array of edges: {('A', 'B'), ('A', 'C'),('A', 'D'), ('A', 'E'), ('A', 'F'), ('A', 'G'), ('B', 'C'), ('B', 'D'), ('B', 'E'), ('B', 'F'), ('B', 'G'), ('C', 'D'), ('C', 'E'), ('C', 'F'), ('C', 'G'), ('D', 'E'), ('D', 'F'), ('D', 'G'), ('E', 'F'), ('E', 'G'), ('F', 'G')}
The result for algorithm Clique will be 2 elements list: {FEDCBAG, H}
The result for algorithm maximumClique will be 1 elements list: {FEDCBAG}

Example 4:
The Graph with the array of nodes = {'A', 'B', 'C', 'D', 'E', 'F'} and an array of edges: {('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'), ('F', 'A'), ('F', 'B'), ('F', 'C'), ('E', 'B'), ('E', 'C')}
The result for algorithm Clique will be 3 elements list: {BAF, CBEF, CDE}
The result for algorithm maximumClique will be 1 elements list: {BAF, CBEF, CDE}

Example 5:
The Graph with the array of nodes = {('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D'), ('C', 'E'), ('E', 'F'), ('E', 'H'), ('E', 'G'), ('F', 'G'), ('G', 'H'), ('H', 'K'), ('H', 'L'), ('K', 'L'), ('K', 'M'), ('I', 'K'), ('J', 'K'), ('J', 'L'), ('I', 'M'), ('J', 'M'), ('I', 'J'), ('J', 'O'), ('O', 'P'), ('N', 'O'), ('P', 'S'), ('N', 'S'), ('S', 'T'), ('S', 'V'), ('N', 'V'), ('N', 'X'), ('T', 'V'), ('V', 'X'), ('P', 'T'), ('Q', 'S'), ('Q', 'P'), ('Q', 'R'), ('P', 'R'), ('R', 'U'), ('U', 'S'), ('U', 'T'), ('C', 'R')}
The result for algorithm Clique will be 19 elements list: {CBAD, CR, CE, GEH, FEG, KHL, JIKM, JO, JKL, VNX, NSV, NO, OP, PQR, QPS, PST, RU, TSU, TSV}
The result for algorithm maximumClique will be 2 elements list: {CBAD, JIKM}

Estimation of algorithm's complexity:

<u>Time complexity</u> – as calculated in the section where algorithm has been presented, the time complexity for pSet would be O(2^n)
*The time complexity for Clique should be: O(n^3) + O(n) (in case input is an complete graph) + O(n*2^n) +O(n^3*6^n)+ O (n^5*2^n + n^3*6^n) (in case the input is not a complete graph) = O (n^5*2^n + n^3*24^(n/3)) = O (n^3*24^(n/3))*
*The time complexity for maximumClique should be: O (n^3*24^(n/3)) + O(3^(n/3)) + O(3^(n/3)) = O (n^3*24^(n/3))*

<u>Space complexity</u> – all three algorithms: pSet, Clique and maximumCliques would create a linked list with extendable size, the pSet would put in result 2^n.
*The Clique would created: n*n for nodes and those nodes that are connected to its, n*2^n for all possible combination of nodes and its children, 3^(n/3) for all possible cliques.*
The third algorithm would use 3^(n/3) for all possible cliques for all possible cliques and k space for maximum cliques (k is depends on the graph)
The space complexity for all 3 algorithms should be O(n) since it would be depends on the input graph (expecially number of vertexes n)