Team Members

يوسف مصطفي عباس القللي : 20191700792
اسراء عبدالنبي خليل مدبولي : 20191700098
تغريد رافت فاروق عبدالغني : 20191700193

Team : T196

IN synsets file:

-We using dictionary as a data structure
to store my information.
-using it because it is easy for using and
can access it easily.

-In Dictionary, key must be unique. Duplicate keys are not allowed if you try to use duplicate key then compiler will throw an exception.
-In Dictionary, you can only store same types of elements.
-The capacity of a Dictionary is the number of elements that Dictionary can hold.

IN hypernyms file:

-We using dictionary as a data structure
to store my information.
-using it because it is easy for using and
can access it easily.

-In Dictionary, key must be unique. Duplicate keys are not allowed if you try to use duplicate key then compiler will throw an exception.
-In Dictionary, you can only store same types of elements.
-The capacity of a Dictionary is the number of elements that Dictionary can hold.

In function shortest common ancestor:

Using algorithm : **BFS**:
This algorithm using to find shortest path because in BFS you can reach a vertex with a minimum number of edge from source to vertex, and using queue as a data structure :that is mean first in first out .
How it working !?
BFS :work:
Algorithm visit and marking starting node ,then its moves towards the nearest unvisited nodes and analysis them ,once visited ,all nodes are marked .you will iterations this steps utill when make sure all nodes and neighbors are visited and marked successfully.

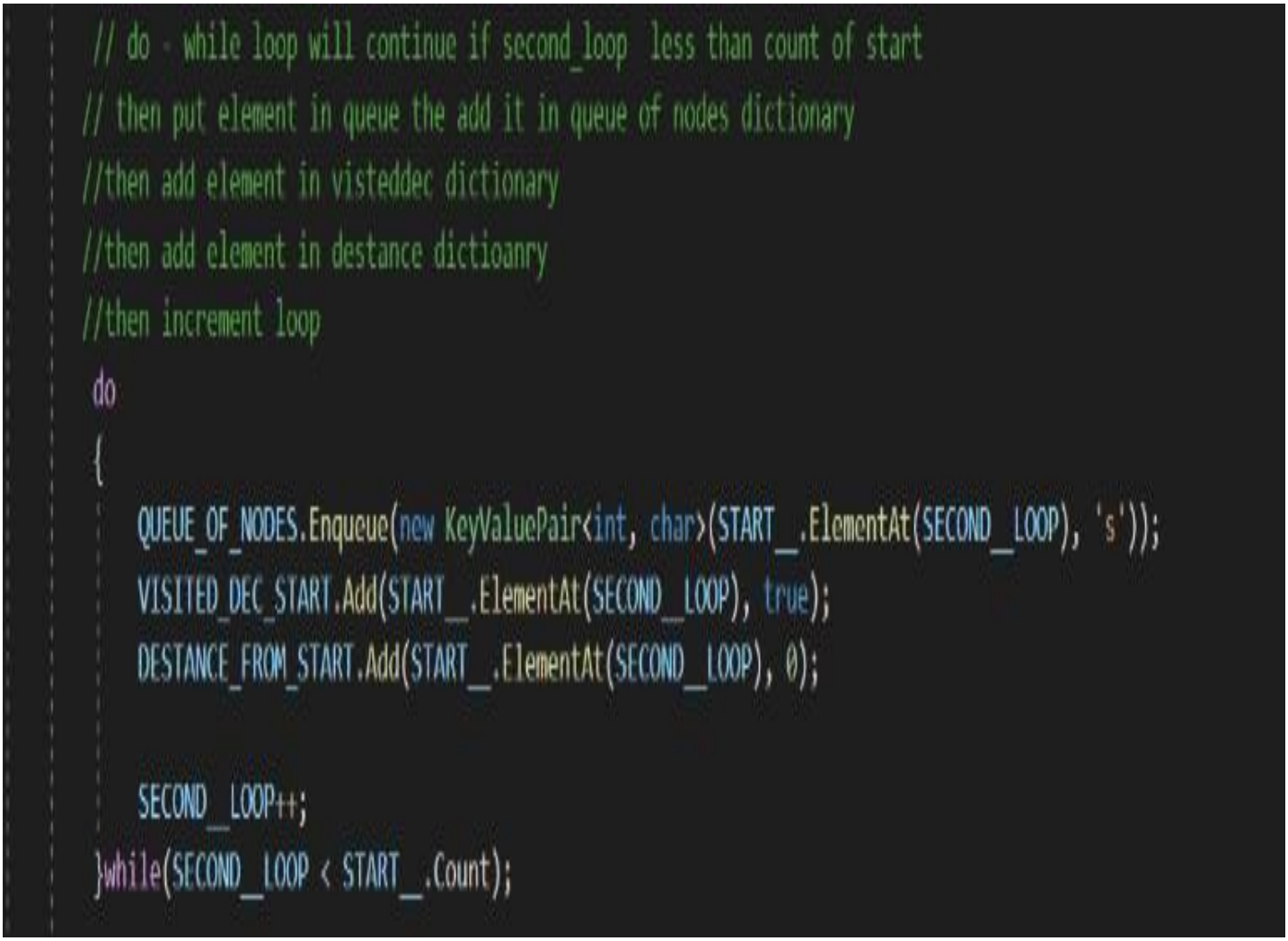
This part will discuss and analysis code of shortest common ancestor with photo :

first :
we define some variables and some dictionary in your code .
then enter to do_while loop to fill and check node has been visited and put them in new dictionary if has vistited and has letter "g".

```
1 reference
public int GET_SHORTEST_COMMEN_ANCESTPRS(string PN1_, string PN2_, out HashSet<string> LIST_OF_SYNSET) // o(v^2)
{
    // this lines o(1)
    int FIRST_LOOP = 0;
    int SECOND__LOOP = 0;
    int THIRD__LOOP = 0;
    var GOAL__ = GRAPH.____NOUNS_GRAPH__[PN2_];
    var START__ = GRAPH.____NOUNS_GRAPH__[PN1_];
    var DESTANCE_FROM_START = new Dictionary<int, int>();
    var DESTANT__TO_GOAL = new Dictionary<int, int>();
    var VISITED_DEC_START = new Dictionary<int, bool>();
    var VISITED_DEC_GOAL = new Dictionary<int, bool>();
    var __MINIMAM_DESTANS = int.MaxValue;
    int SHORTEST_COMMEN_ANCESTPRS = 0;
    var QUEUE_OF_NODES = new Queue<KeyValuePair<int, char>>();

    //a char is whether an 's' or an 'g' to see where that node comes from in the current front of the queue
    do // o(v)
    {
        VISITED_DEC_GOAL.Add(GOAL__.ElementAt(FIRST_LOOP), true);
        QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(GOAL__.ElementAt(FIRST_LOOP), 'g'));
        DESTANT__TO_GOAL.Add(GOAL__.ElementAt(FIRST_LOOP), 0);

        FIRST_LOOP++;
    } while (FIRST_LOOP < GOAL__.Count);
    do // o(v)
    {
        QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(START__.ElementAt(SECOND__LOOP), 's'));
        VISITED_DEC_START.Add(START__.ElementAt(SECOND__LOOP), true);
        DESTANCE_FROM_START.Add(START__.ElementAt(SECOND__LOOP), 0);

        SECOND__LOOP++;
    }while(SECOND__LOOP < START__.Count);
```

Second do_while loop :

That continue and full start dictionary that has node with letter "s",then add it in distance dictionary.



```
// do - while loop will continue if second_loop  less than count of start
// then put element in queue the add it in queue of nodes dictionary
//then add element in visteddec dictionary
//then add element in destance dictioanry
//then increment loop
do
{
    QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(START_.ElementAt(SECOND_LOOP), 's'));
    VISITED_DEC_START.Add(START_.ElementAt(SECOND_LOOP), true);
    DESTANCE_FROM_START.Add(START_.ElementAt(SECOND_LOOP), 0);

    SECOND_LOOP++;
}while(SECOND_LOOP < START_.Count);
```

Threed thing :
Will enter to while loop to check somethings :
First we store first element of queue_of _nodes in queue_front and not delete it
Then check condition value of queue is visited and start or visited and goal (your node)
Then calculate distance from value of dictionary
After that compare the minimum distance and save it .

```
//traverse the graph !wisely
while(QUEUE_OF_NODES.Count > 0)
{
    //check this instriction
    //if it's a start node and  visited in goals nodes
    //if it's a goal node and  visited in start nodes
    var queueFront = QUEUE_OF_NODES.Peek();
    if ((queueFront.Value == 'g' && VISITED_DEC_START.ContainsKey(queueFront.Key)) ||
        (queueFront.Value == 's' && VISITED_DEC_GOAL.ContainsKey(queueFront.Key)))
    {
        var currentDistance = 0 + DESTANT__TO_GOAL[queueFront.Key] + DESTANCE_FROM_START[queueFront.Key]
         if(currentDistance < _MINIMAM_DESTANS)
        {
            SHORTEST_COMMEN_ANCESTPRS = QUEUE_OF_NODES.Peek().Key;
            //store the first element in queue in sortest comman without delete it
            _MINIMAM_DESTANS = currentDistance;//store currnet distancr in min des
        }
    }
```

Four thing :
For loop to to check two things :
First condition:
If node has start in queue and check if node has visited in your queue in dictionary :if not visited add it in started queue and visit it this loop is cycle otherwise break.
Second condirion :
If node has your goal (your node) in queue and check if node has visited in your queue in dictionary :if not visited add it in goal queue and visit it this loop is cycle otherwise break.

```
for (var i = 0; i < GRAPH[queueFront.Key].Count; i++)
{
    var node = GRAPH[queueFront.Key].ElementAt(i);
        //check if value of node equal to 's'
    if (queueFront.Value == 's')
    {       //check if node is visted
            //if not then add in dictionary that is visited
            //then add in queue node that has letter 's'
            // then add this node in distance dictionary

        if (!VISITED_DEC_START.ContainsKey(node))
        {
                VISITED_DEC_START.Add(node, true);
                QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(node, 's'));

                DESTANCE_FROM_START.Add(node, DESTANCE_FROM_START[queueFront.Key] + 1);
        }
    }
    else if (queueFront.Value == 'g')
    {
        if (!VISITED_DEC_GOAL.ContainsKey(node))
        {
                //check if node is visted
                //if not then add in dictionary that is visited
                //then add in queue node that has letter 'g'
                // then add this node in distance dictionary

                VISITED_DEC_GOAL.Add(node, true);
                QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(node, 'g'));

                DESTANT__TO_GOAL.Add(node, DESTANT__TO_GOAL[queueFront.Key] + 1);
        }
    }
}
    QUEUE_OF_NODES.Dequeue();
}
LIST_OF_SYNSET = GRAPH.__SYNSET_GRAPH__[SHORTEST_COMMEN_ANCESTPRS];
return __MINIMAM_DESTANS;
```

This function will check the nodes is visited or not and calucuate the shortest distance to reach to your goal

```
//traverse the graph twisely
while(QUEUE_OF_NODES.Count > 0) // o(v^2)
{
    var queueFront = QUEUE_OF_NODES.Peek(); //o(1)
    if ((queueFront.Value == 'g' && VISITED_DEC_START.ContainsKey(queueFront.Key)) ||
        (queueFront.Value == 's' && VISITED_DEC_GOAL.ContainsKey(queueFront.Key))) //o(1)
    {
        var currentDistance = 0 + DESTANT__TO_GOAL[queueFront.Key] + DESTANCE_FROM_START[queueFront.Key] ;
         if(currentDistance < __MINIMAM_DESTANS) // o(1)
        {
            SHORTEST_COMMEN_ANCESTPRS = QUEUE_OF_NODES.Peek().Key;
            __MINIMAM_DESTANS = currentDistance;
        }
    }

    for (var i = 0; i < GRAPH[queueFront.Key].Count; i++) // o(v)
    {
        var node = GRAPH[queueFront.Key].ElementAt(i);

        if (queueFront.Value == 's') // o(1)
        {
            if (!VISITED_DEC_START.ContainsKey(node)) // o(1)
            {
                    VISITED_DEC_START.Add(node, true);
                    QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(node, 's'));

                    DESTANCE_FROM_START.Add(node, DESTANCE_FROM_START[queueFront.Key] + 1);
            }
        }
        else if (queueFront.Value == 'g') // o(1)
        {
            if (!VISITED_DEC_GOAL.ContainsKey(node)) // o(1)
            {
                    VISITED_DEC_GOAL.Add(node, true);
                    QUEUE_OF_NODES.Enqueue(new KeyValuePair<int, char>(node, 'g'));

                    DESTANT__TO_GOAL.Add(node, DESTANT__TO_GOAL[queueFront.Key] + 1);
            }
        }
    }
    QUEUE_OF_NODES.Dequeue();
}
LIST_OF_SYNSET = GRAPH.__SYNSET_GRAPH__[SHORTEST_COMMEN_ANCESTPRS];
return __MINIMAM_DESTANS;
}
```

This function to check the root and if this node belongs to this parent or not in your graph

```
1 reference
public bool CHECK__THE_ROOTE() // o(v^2)
{
    //this lines o(1)
    int __ROOTED = 0;
    int COUNT_OF_ROOT = 0;
    bool __ASSIGNED = false;
    int COUNT = 0;
    var VALUE__OF_KEY = GRAPH.__Graph__.Keys.ElementAt(0);
    int IN_COUNT = 0;
    var PARENT__ = GRAPH[VALUE__OF_KEY].ElementAt(0);

    while (COUNT < GRAPH.__Graph__.Keys.Count()) // o(v^2)
    {
        VALUE__OF_KEY = GRAPH.__Graph__.Keys.ElementAt(COUNT);
        while(IN_COUNT < GRAPH[VALUE__OF_KEY].Count) // o(v)
        {
            PARENT__ = GRAPH[VALUE__OF_KEY].ElementAt(IN_COUNT);
            if (GRAPH[PARENT__].Count == 0) //o(1)
            {
                if (__ASSIGNED) //o(1)
                {
                    if (__ROOTED != PARENT__) //o(1)
                    {
                        COUNT_OF_ROOT++;
                        __ROOTED = PARENT__;
                    }
                }
                else //o(1)
                {
                    __ROOTED = PARENT__;
                    COUNT_OF_ROOT++;
                    __ASSIGNED = true;

                }
            }
            IN_COUNT++;
        }
        COUNT++;
    }
    return COUNT_OF_ROOT == 1;
}
```

This function will read all your data in your files and spilt it to more one things such as ID,name .

```
1 reference
private void Initialize__NOUNS__(string pFilePath) // o(v^2 log(n))
{
    // this variable o(1)
    var __READER_ = new StreamReader(pFilePath);
    string __LINES;
    string[] DATA__;
    int NOUN__ID;
    //o(v^2 log(n))
    do
    {
        // this line o(1)
        __LINES = __READER_.ReadLine();
        DATA__ = __LINES.Split(",");
        string[] NOUNS__ = DATA__[1].Split(' ');
        int.TryParse(DATA__[0], out NOUN__ID);
        __SYNSET_GRAPH__.Add(NOUN__ID, new HashSet<string>(NOUNS__));

        for (int i = 0; i < NOUNS__.Count(); i++) // o(v log(n))
        {
            SortedSet<int> nounIds;
            if (!____NOUNS_GRAPH__.TryGetValue(NOUNS__[i], out nounIds)) //o(1)
            {
                nounIds = new SortedSet<int>();
            }
            nounIds.Add(NOUN__ID); //o(log(n))
            if (____NOUNS_GRAPH__.ContainsKey(NOUNS__[i])) // o(1)
            {
                ____NOUNS_GRAPH__[NOUNS__[i]] = nounIds;
            }

            else // o(1)
            {
                ____NOUNS_GRAPH__.Add(NOUNS__[i], nounIds);
            }


        }

    } while (!__READER_.EndOfStream);
}
```

this function make to draw your grapth from your data to compare to another file to check nodes that belongs to owner parent and write correct data.

```
1 reference
private void DRAW_GRAPH_HYPERNAMES(string Hypernyms__File) //o(v^2)
{
    //this lines o(1)
    var __READER_ = new StreamReader(Hypernyms__File);
    string __LINES;
    string[] strs_plit1;
    string CHILD;
    int INT__CHILD;

    do //o(v^2)
    {
        //this lines o(1)
        __LINES = __READER_.ReadLine();
        strs_plit1 = __LINES.Split(',');
        CHILD = strs_plit1[0];
        var LIST_TO_CHILD = new HashSet<int>();
        int i = 1;

        while(i < strs_plit1.Count()) // o(n)
        {
            LIST_TO_CHILD.Add(int.Parse(strs_plit1[i]));
            i++;
        }

        INT__CHILD = int.Parse(CHILD); // o(1)

        if (!__Graph__.ContainsKey(INT__CHILD)) //o(1)
        {
            __Graph__.Add(int.Parse(CHILD), LIST_TO_CHILD);
        }

    } while (!__READER_.EndOfStream);

}
```

```
1 reference
public HashSet<int> GET__PARENTS_TO_SPACIFIC_CHILD(int child) //o(1)
{
    if (__Graph__.ContainsKey(child)) //o(1)
        return __Graph__[child];

    return new HashSet<int>();
}
6 references
public HashSet<int> this[int key] => GET__PARENTS_TO_SPACIFIC_CHILD(key); //o(1)
```

Put your data to check shortest path from another function and calculate realations between list of nodes

```
public int GetSca(string s1, string s2, out HashSet<string> scList) // o(v^2)
{
    return DATA__PROC.GET_SHORTEST_COMMEN_ANCESTPRS(s1, s2, out scList); // o(v^2)
}
1 reference
int SemanticRelation(string s1, string s2)// o(v^2)
{
    HashSet<string> n;
    return GetSca(s1, s2, out n);// o(v^2)
}
1 reference
```

```
public string PRINT__OUTCAST__NOUN(List<string> NOUNS)// o(v^4)
{
    //this lines o(1)
    int MAX = 0;
    string OUTCAST__ = "all equal";
    var COMPARSION = true;

    for(int v = 0;v< NOUNS.Count();v++) // o(v^4)
    {
        int SUM = 0;
        for(int j=0;j< NOUNS.Count();j++) // o(v^3)
        {
            SUM += SemanticRelation(NOUNS[v], NOUNS[j]); // o(v^2)
        }
        if (!COMPARSION) // o(1)
        {
            if (SUM >= MAX) // o(1)
            {
                MAX = SUM;
                OUTCAST__ = NOUNS[v];
            }
        }
        else // o(1)
        {
            MAX = SUM;
            COMPARSION = false;
            OUTCAST__ = NOUNS[v];
        }
    }
    return OUTCAST__;
}
```

This is the main function that :

You can read your selected fils from your pc and compare your files to graph and reations to check if correct or not and put results in new files to easy to compare your new files to main files in your test cases.

```
static void Main(string[] args)
{
    //path of each file
    string[] lines1 = System.IO.File.ReadAllLines("Testcases/Sample/Case1/Input/3RelationsQueries.txt");
    string[] lines2 = System.IO.File.ReadAllLines("Testcases/Sample/Case1/Input/4OutcastQueries.txt");
    WORD_NET DataGraph = new WORD_NET("Testcases/Sample/Case1/Input/2hypernyms.txt", "Testcases/Sample/Case1/Input/1synsets.txt");
    string[] output_relation = new string[lines1.Length];//read relation file line by line in output_relation array
    string[] output_outcast = new string[lines2.Length];//read outcast file line by line in output_outcast array
    // next four variable o(1)
    HashSet<String> hs;
    string[] strs_plit1;
    int output;
    string concatenate_result;

    for (int i = 1; i< lines1.Length;i++) // o(v^3)
    {
        concatenate_result = "";
        strs_plit1 = lines1[i].Split(",");
        output = DataGraph.GetSca(strs_plit1[0], strs_plit1[1], out hs);// o(v^2)
        concatenate_result += output + "," ;
        foreach (var x in hs) // o(v)
        {
            concatenate_result += x;
        }
        output_relation[i - 1] = concatenate_result;
    }
```