CS301

2022-2023 Spring

Project Final Report

Group 017

Sezin Tekin 28884

Alkım Özyüzer 29263

27/05/2023

1. Problem Description

Our problem is about Degree Constrained Spanning Tree with an input of n-node undirected graph G(V, E); positive integer k <= n, and our question is “What is the smallest integer k such that G has a spanning tree in which no node has degree greater than k?”. A degree-constrained spanning tree in graph theory is one whose maximum vertex degree is constrained to a fixed constant k. Degree Constrained Spanning Trees can be used to design communication networks. It requires connecting a set of nodes in a network such that each node has a finite number of connections, and this can be modeled as a DCST problem as the degree of each node represents the maximum number of connections it can have.

The problem at hand is to determine the smallest integer k such that an undirected graph G (V, E) has a spanning tree in which no node has a degree greater than k. In other words, we need to find the maximum degree of any node in a spanning tree of G. The goal is to determine whether such a spanning tree exists and to identify the smallest value of k that satisfies this constraint. The value of k must be a positive integer less than or equal to the number of vertices in G. This problem is known as the degree-constrained spanning tree problem.

Theorem:
The Degree Constrained Spanning Tree (DCST) problem is NP-complete.
Proof:
To prove that DCST is NP-complete, we need to show that it is in NP class and is NP-hard by reducing a known NP-complete problem.

1. DCST is in NP:
Given a spanning tree input that spans the T tree, we can check if it contains all the vertices in V and whether the degree of each vertex is at most k, and this validation can be done in O(|V|) time, so DCST is in NP since it takes polynomial time.

2. DCST is NP-hard:
We shall reduce the Hamilton Path problem to DCST in order to demonstrate this. Polynomial reducibility is defined in Richard M. Karp's 1975 book Reducibility Among Combinatorial Problems, which also asserts the requirement that a language L is NP-complete if it is in NP and any language in NP can be polynomially reducible to L. We shall show that the Hamilton

path problem, a well-known NP-complete problem, may be reduced to the DCST problem in order to show that the DCST problem is in NP and that it is NP-hard.

Finding a simple path that traverses every vertex on a given graph exactly once is the goal of the graph theory problem known as the Hamilton Path. This problem is an NP-complete problem that inquires as to whether such a path exists in each graph. The Degree Constrained Spanning Tree (DCST) problem and the Hamilton Path problem are related because the DCST problem can be used to show that the Hamilton Path problem is NP-hard.The connection between the Hamilton Path problem and the Degree Constrained Spanning Tree (DCST) problem is that one can be reduced to the other by demonstrating the NP-hardness of the DCST problem.

In the Hamilton Path problem, we are looking for a simple path that visits each vertex exactly once, and in the DCST problem, we are looking for a spanning tree with the degrees of each vertex bounded by a certain integer k.

We will reduce the Hamiltonian Path problem to DCST as follows:

From the G graph we have, we will create a new G' graph and this graph will be the same as the G graph. Since we want to find a path that visits each vertex exactly once, we will choose k as 2.

If the graph G' has a Hamiltonian Path, then there is a path that visits each vertex exactly once. This path is a spanning tree G' with a degree constraint of 2 since every vertex except the start and end vertices has 2 degrees and the start and end vertices have degrees 1.

If G' has a DCST with degree constraint k=2, then G' has a spanning tree T in which each vertex has at most 2 degrees. Since G' and G have the same vertices and sides, T is also in G. Since T is a spanning tree with each vertex at most 2 degrees, there must be a path that visits each vertex exactly once. Therefore, G has a Hamiltonian Path.

The DCST problem is NP-hard since we can reduce the Hamiltonian Path problem to the Degree Constrained Spanning Tree problem in polynomial time.

2. Algorithm Description

a. Brute Force Algorithm

Brute Force Algorithm is an approach that finds all possible choices until a solution is found and its time complexity is proportional to the input size. For our Degree Constrained Spanning Tree problem, we can also use the Brute Force Algorithm by generating all the possible spanning trees of our given Graph G (V, E). Then checking if any of these trees satisfy the degree constraint k. The algorithm takes our graph G as the inputs and then generates all the possible spanning trees with a function. The function for generating all the possible trees can be implemented using depth-first search-based backtracking. And then, we iterate through all the spanning trees to check if the tree satisfies the degree constraint. This checking iterations will be done by another function. The algorithm continues to iterate over the next tree until the degree constraint condition will be satisfied and when it is satisfied, that appropriate k value will be returned. If there is no solution found, then the algorithm returns None. Which means that there is no such k value that satisfies our condition.

The Pseudocode for the Algorithm:

```
implement a function that generates all possible spanning trees;
implement a bool function that checks the degree constraint:
        for node in tree.nodes:
            if node.degree > k:
                return False
        return True

implement the Brute Force Algorithm function:
        all_spanning_trees = call the function for generating all possible trees
            for tree in all_spanning_trees:
                if (call the function that checks the degree constraint):
                    return k value
            return -1
```

The best-case scenario happens when the degree constraint is satisfied by the first spanning tree that is produced. As there would only be one tree to construct and check in this scenario, the algorithm's time complexity would be O(1). But the worst-case scenario occurs when none of the generated spanning trees satisfy the degree constraint. The worst-case complexity of the algorithm for the DCST problem is exponential since there are n ^ (n - 2) possible spanning trees for the graph G which has n vertices, and the algorithm iterates through all the spanning trees in a worst case scenario. As we said earlier, since the complexity of the brute Force Algortihm's complexity is proportional to the input size, as the tree gets bigger the time complexity also increases. So, this algorithm can be considered as efficient for small spanning trees.

b. Heuristic Algorithm

A heuristic algorithm is a problem-solving approach that uses practical rules or techniques to find an approximate solution rather than an optimal solution.

For our solution, a heuristic approach is used to find the smallest value of k for which a degree-constrained spanning tree exists in the given graph. The algorithm starts by sorting the edges of the graph based on their weights. Then, it iterates through the sorted edges and checks if an edge can be added to the minimum spanning tree (MST) while satisfying the degree constraint (using the can_add_edge function). If an edge can be added, it is included in the MST, and the corresponding vertices are connected in the adjacency list.

After constructing the MST, the algorithm checks if all nodes are connected by finding the root of the disjoint sets. If all nodes have the same root, it means the constructed graph is a spanning tree. If not, it means not all nodes are connected, and there is no spanning tree.

The heuristic algorithm consists of two functions: heuristicalgorithm(G, k) and find_smallest_k(G).

The degree constraint k and a graph G are the inputs for the heuristic algorithm function. Based on their weights, it arranges the edges of G in ascending order. Then, to keep track of the sets of vertices, it builds an empty graph MST and a disjoint set data structure ds. The connections in MST are stored in an empty adjacency list adj. When checking if an edge may be added to MST while satisfying the degree constraint, the algorithm iterates through the sorted edges. If so, it changes the sets in ds, the adjacency list adj, and adds the edge to MST.

Then, by locating the set's root in ds, it determines if all nodes are connected in MST. The function returns true if a degree-constrained spanning tree exists and all nodes share the same root. Otherwise, false is returned.

The input for the find_smallest_k function is a graph G. It sets the right variable to the number of vertices in G and the left variable to 1. To determine the minimum value of k for which a degree-constrained spanning tree exists in G, it conducts a binary search. By invoking the heuristicalgorithm function, it determines the midpoint between left and right and determines whether a degree-constrained spanning tree is present for that midpoint value.

It modifies the left or right variable depending on the outcome to reduce the search space. The left value is returned as the lowest integer k satisfying the criterion after the binary search is completed and left is no longer less than right.

3. Algorithm Analysis

a. Brute Force Algorithm

Theorem: The Brute Force Algorithm for the Degree Constrained Spanning Tree problem correctly finds the smallest integer k such that G has a spanning tree in which no node has degree greater than k, if it exists in the input graph G (V, E), and returns None otherwise.

Proof: The algorithm generates all potential spanning trees of G using a depth-first search-based backtracking approach, then checks the degree constraint for each one. Because it employs a depth-first search-based backtracking approach, the function for generating all possible spanning trees will generate all spanning trees of G. Therefore, there will be no spanning tree of G that the algorithm will miss. The bool function that checks the degree constraint returns the smallest number k such that G has a spanning tree with no nodes having a degree larger than k. As a result, if the input graph has a spanning tree that meets the degree criteria, the function will return the lowest such k, else it will return None. Using the bool function, the Brute Force Algorithm function iterates through all potential spanning trees of G and tests if each one satisfies the degree condition. It returns the smallest such k if it finds a spanning tree that meets the degree limit. It returns None if none of the trees satisfy the degree condition. Therefore, if a spanning tree meets the degree criteria, the method will return the smallest such k; otherwise, it will return None. As a result, the Brute Force Algorithm properly finds the smallest integer k such that G has a spanning tree with no node having a degree larger than k.

The worst-case time complexity of the Brute Force Algorithm for the Degree Constrained Spanning Tree problem is Θ(n^(n-1)), where n is the number of vertices in the input graph G (V, E). The reason behind this is that for an input network with n vertices, there are n ^ (n - 2) potential spanning trees. Since each spanning tree has at most (n-1) edges, there are at most n ^ (n - 2) possible spanning trees. As a result, the algorithm must examine the degree constraint for (n-1)! * n ^ (n - 2) possible spanning trees, which is exponential in n. Because the algorithm must generate and check all possible spanning trees, the lower bound of the time complexity is also Ω(n^(n-1)). Thus, the worst-case time complexity of the Brute Force Algorithm for finding the smallest integer k such that G has a spanning tree in which no node has degree greater than k is Θ(n ^ (n - 2)).

b. Heuristic Algorithm

Theorem: For a given graph G(V, E), our heuristic algorithm finds the smallest integer k for which G has a spanning tree in which no node has degree greater than k, if such a k exists.

Proof: The described algorithm works by conducting a binary search over the possible degree restrictions (k-values), which range from 1 to the total number of vertices in the graph. The algorithm tries to build a spanning tree of the graph that complies with the degree restriction for each possible k. By iterating over the graph's edges, sorted by weight, and tentatively including each edge in the tree if it does not break the degree requirement, this construction is accomplished. Each effective inclusion unifies the linked elements of the two edge endpoints, which are controlled by a disjoint-set data structure.

The initial k for which the algorithm successfully builds a spanning tree is guaranteed to be the smallest such k, if it exists, because the binary search starts from the smallest conceivable value of k and grows gradually. As a result, the proposed approach will undoubtedly discover the least degree constraint k for which a degree-constrained spanning tree is viable for the graph.

Next, we need to consider the time complexity. The overall complexity of the algorithm is O(E logE + E + V) . Because sorting the edges takes O(E logE) time and constructing the minimum spanning tree using the sorted edges takes O(E) time. Also checking if all nodes are connected takes O(V) time.

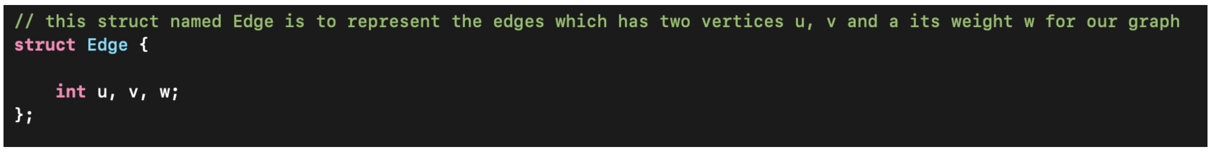
And the space complexity of this algorithm is mainly determined by three data structures:

1. The adjacency list representation of the graph, which typically requires O(V + E) space, where V is the number of vertices and E is the number of edges.
2. The DisjointSet data structure. This data structure, used for the union-find operations, requires O(V) space because it holds a representation for each vertex in the graph.
3. The sorted list of edges, which requires O(E) space.

So, the overall space complexity of the algorithm is O(V + E).

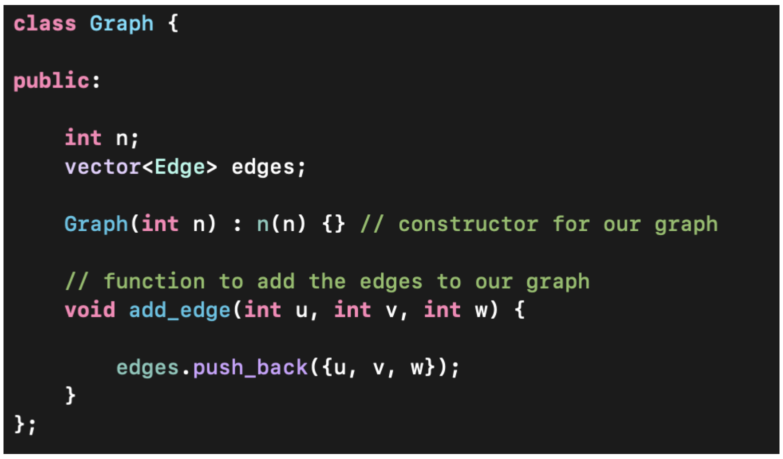
4. Sample Generation (Random Instance Generator)

We need a code that generates random graphs. In the implementation of this code, a struct called Edge and two vertices named u, v inside this struct and a w variable representing the weight of the edge were defined.



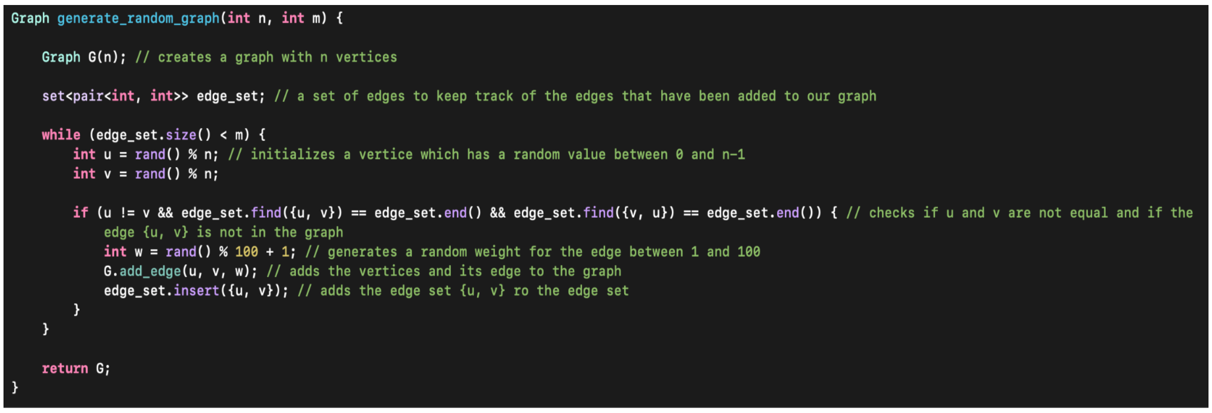
```
// this struct named Edge is to represent the edges which has two vertices u, v and a its weight w for our graph
struct Edge {

    int u, v, w;
};
```

Then, by creating a class called Graph, an int n variable was created to specify n vertices in the graph and an edge vector was created to hold the edges. And a function called **add_edge** was implemented to keep the properties of all edges, and in this function, the vertices and weight variables were taken and pushed into the edge vector.



```cpp
class Graph {

public:

    int n;
    vector<Edge> edges;

    Graph(int n) : n(n) {} // constructor for our graph

    // function to add the edges to our graph
    void add_edge(int u, int v, int w) {

        edges.push_back({u, v, w});
    }
};
```

Later, a function named **generate_random_graph** was created, and this function takes n, and m variables as parameters. N represents the vertice number and m is the total number of edges. This function first creates a Graph G object and creates a set called edge_set to prevent the added edges from being added again. This function adds the edge created by the add_edge function to our G object by creating u and v vertices with random values as long as the number of edges in edge_set is less than m and defining a weight to these vertices. When the size of our edge_set is equal to m, our Graph G is completed and finally, the function will return our Graph.



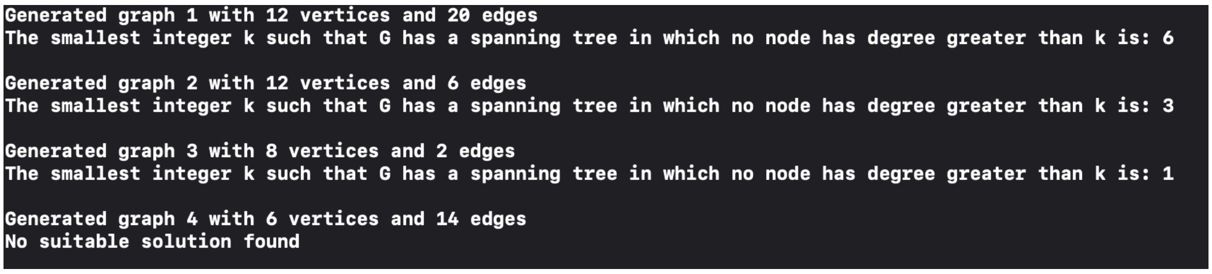
```cpp
Graph generate_random_graph(int n, int m) {

    Graph G(n); // creates a graph with n vertices

    set<pair<int, int>> edge_set; // a set of edges to keep track of the edges that have been added to our graph

    while (edge_set.size() < m) {
        int u = rand() % n; // initializes a vertice which has a random value between 0 and n-1
        int v = rand() % n;

        if (u != v && edge_set.find({u, v}) == edge_set.end() && edge_set.find({v, u}) == edge_set.end()) { // checks if u and v are not equal and if the
            edge {u, v} is not in the graph
            int w = rand() % 100 + 1; // generates a random weight for the edge between 1 and 100
            G.add_edge(u, v, w); // adds the vertices and its edge to the graph
            edge_set.insert({u, v}); // adds the edge set {u, v} ro the edge set
        }
    }

    return G;
}
```

5. Algorithm Implementations

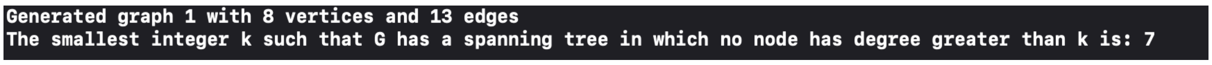
a. Brute Force Algorithm

After completing the implementation of the necessary functions, we also wrote the **brute_force_algorithm** and generated 15 random graphs in our code to search for the minimum k values one by one. However, the time complexity of the code is quite high as

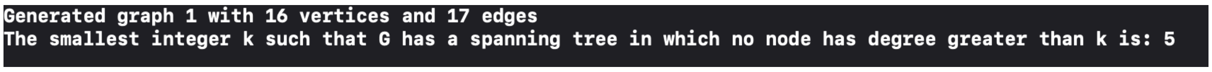
the Depth First Search (DFS) approach is applied to construct all possible spanning trees that are part of the code. As noted above, generating all the possible spanning trees takes exponential time in its worst case, so it took a long time on our computers to finish running the code. For this reason, we created a total of 15 graphs by executing and rerunning the code several times and reached our result.
Here are the results:



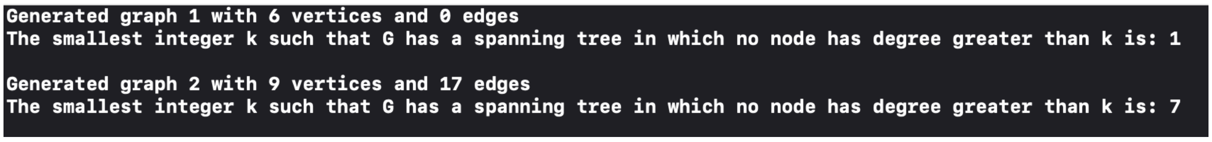
```
Generated graph 1 with 12 vertices and 20 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 6

Generated graph 2 with 12 vertices and 6 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 3

Generated graph 3 with 8 vertices and 2 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 1

Generated graph 4 with 6 vertices and 14 edges
No suitable solution found
```



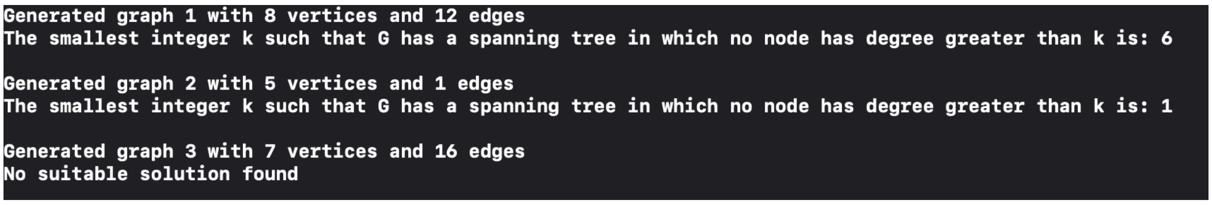
```
Generated graph 1 with 8 vertices and 13 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 7
```



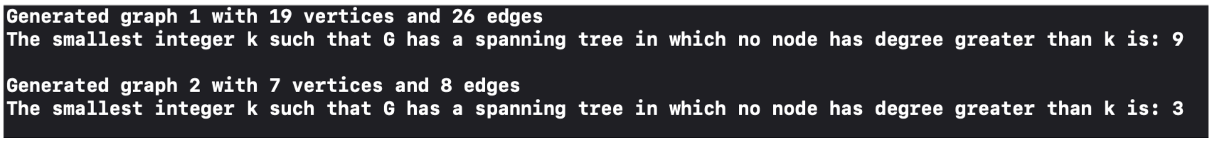
```
Generated graph 1 with 16 vertices and 17 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 5
```



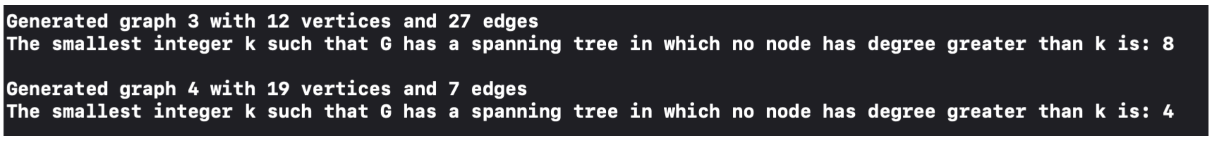
```
Generated graph 1 with 6 vertices and 0 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 1

Generated graph 2 with 9 vertices and 17 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 7
```



```
Generated graph 1 with 8 vertices and 12 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 6

Generated graph 2 with 5 vertices and 1 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 1

Generated graph 3 with 7 vertices and 16 edges
No suitable solution found
```

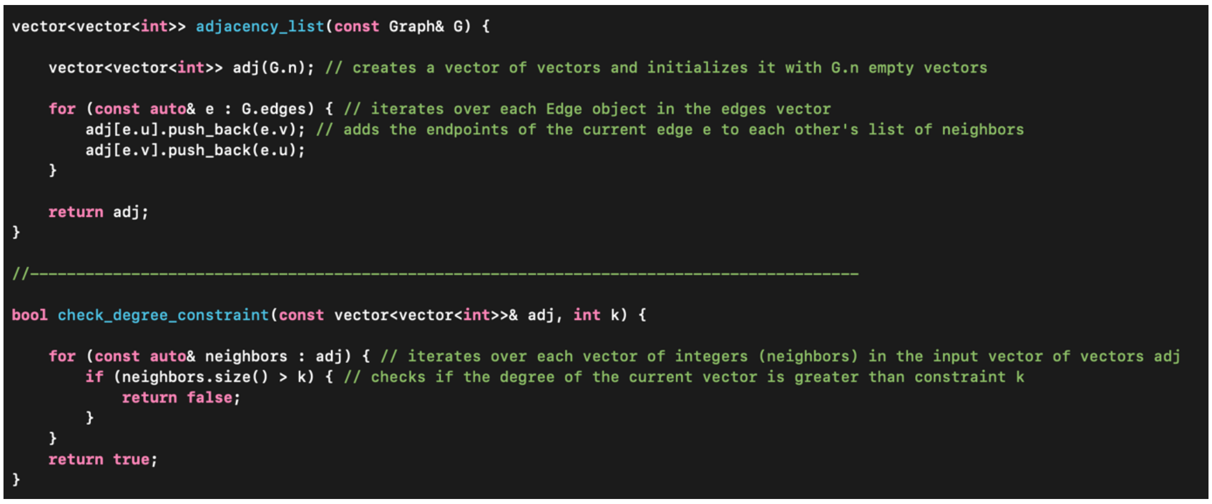


```
Generated graph 1 with 19 vertices and 26 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 9

Generated graph 2 with 7 vertices and 8 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 3
```

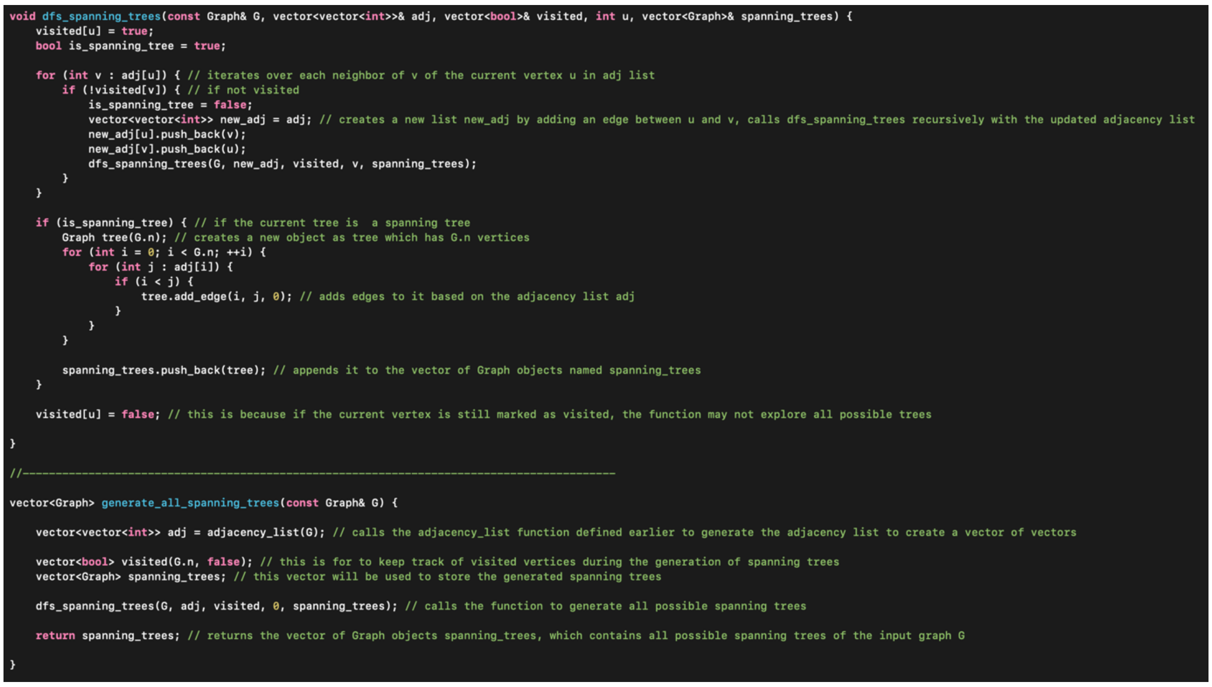


```
Generated graph 3 with 12 vertices and 27 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 8

Generated graph 4 with 19 vertices and 7 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 4
```

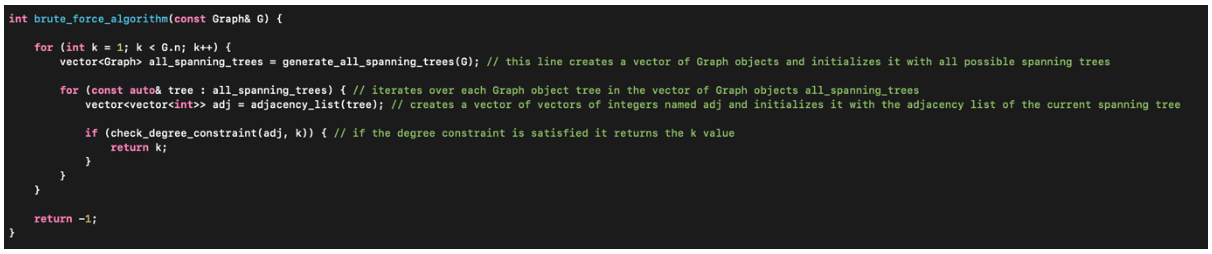
When the results of the code were examined, the smallest k value was found for 13 graphs among the 15 random graphs that provided the constraint of the degrees of the nodes of a graph. However, the minimum k value could not be found for 2 graphs. This may be because any of the all sample trees of the graph have a degree greater than k for any node. The functions and the main part of the codes are given below:



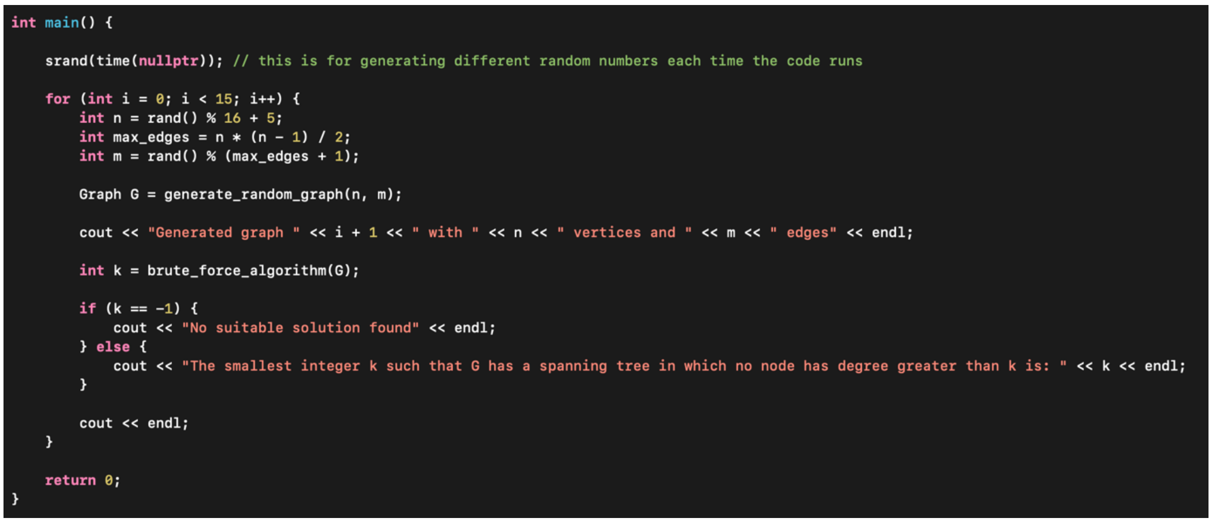
```cpp
vector<vector<int>> adjacency_list(const Graph& G) {

    vector<vector<int>> adj(G.n); // creates a vector of vectors and initializes it with G.n empty vectors

    for (const auto& e : G.edges) { // iterates over each Edge object in the edges vector
        adj[e.u].push_back(e.v); // adds the endpoints of the current edge e to each other's list of neighbors
        adj[e.v].push_back(e.u);
    }

    return adj;
}

//-------------------------------------------------------------------------------------------

bool check_degree_constraint(const vector<vector<int>>& adj, int k) {

    for (const auto& neighbors : adj) { // iterates over each vector of integers (neighbors) in the input vector of vectors adj
        if (neighbors.size() > k) { // checks if the degree of the current vector is greater than constraint k
            return false;
        }
    }
    return true;
}
```



```cpp
void dfs_spanning_trees(const Graph& G, vector<vector<int>>& adj, vector<bool>& visited, int u, vector<Graph>& spanning_trees) {
    visited[u] = true;
    bool is_spanning_tree = true;

    for (int v : adj[u]) { // iterates over each neighbor of v of the current vertex u in adj list
        if (!visited[v]) { // if not visited
            is_spanning_tree = false;
            vector<vector<int>> new_adj = adj; // creates a new list new_adj by adding an edge between u and v, calls dfs_spanning_trees recursively with the updated adjacency list
            new_adj[u].push_back(v);
            new_adj[v].push_back(u);
            dfs_spanning_trees(G, new_adj, visited, v, spanning_trees);
        }
    }

    if (is_spanning_tree) { // if the current tree is  a spanning tree
        Graph tree(G.n); // creates a new object as tree which has G.n vertices
        for (int i = 0; i < G.n; ++i) {
            for (int j : adj[i]) {
                if (i < j) {
                    tree.add_edge(i, j, 0); // adds edges to it based on the adjacency list adj
                }
            }
        }

        spanning_trees.push_back(tree); // appends it to the vector of Graph objects named spanning_trees
    }

    visited[u] = false; // this is because if the current vertex is still marked as visited, the function may not explore all possible trees

}

//-------------------------------------------------------------------------------------------

vector<Graph> generate_all_spanning_trees(const Graph& G) {

    vector<vector<int>> adj = adjacency_list(G); // calls the adjacency_list function defined earlier to generate the adjacency list to create a vector of vectors

    vector<bool> visited(G.n, false); // this is for to keep track of visited vertices during the generation of spanning trees
    vector<Graph> spanning_trees; // this vector will be used to store the generated spanning trees

    dfs_spanning_trees(G, adj, visited, 0, spanning_trees); // calls the function to generate all possible spanning trees

    return spanning_trees; // returns the vector of Graph objects spanning_trees, which contains all possible spanning trees of the input graph G

}
```



```cpp
int brute_force_algorithm(const Graph& G) {

    for (int k = 1; k < G.n; k++) {
        vector<Graph> all_spanning_trees = generate_all_spanning_trees(G); // this line creates a vector of Graph objects and initializes it with all possible spanning trees

        for (const auto& tree : all_spanning_trees) { // iterates over each Graph object tree in the vector of Graph objects all_spanning_trees
            vector<vector<int>> adj = adjacency_list(tree); // creates a vector of vectors of integers named adj and initializes it with the adjacency list of the current spanning tree

            if (check_degree_constraint(adj, k)) { // if the degree constraint is satisfied it returns the k value
                return k;
            }
        }
    }

    return -1;
}
```



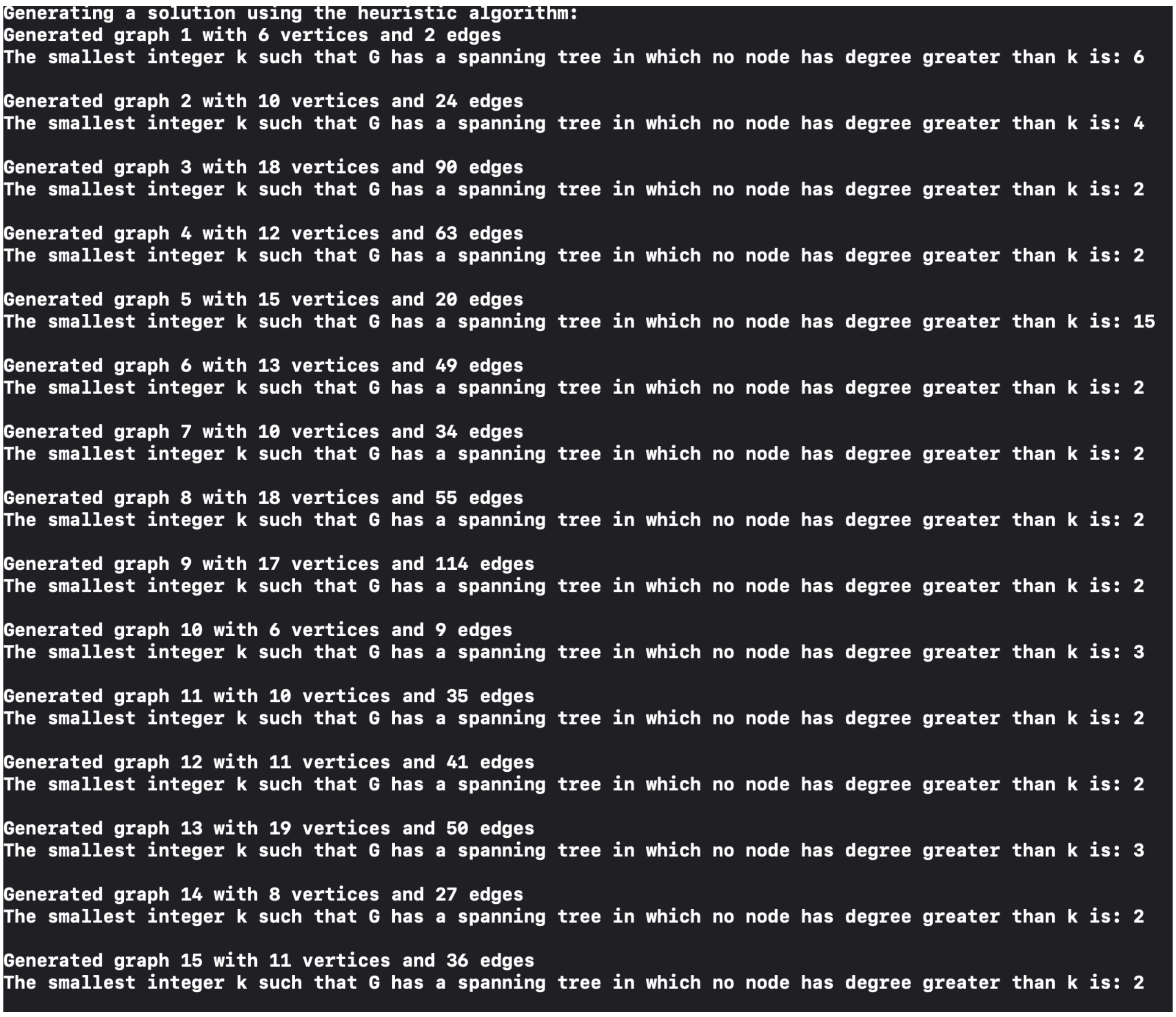
```
int main() {

    srand(time(nullptr)); // this is for generating different random numbers each time the code runs

    for (int i = 0; i < 15; i++) {
        int n = rand() % 16 + 5;
        int max_edges = n * (n - 1) / 2;
        int m = rand() % (max_edges + 1);

        Graph G = generate_random_graph(n, m);

        cout << "Generated graph " << i + 1 << " with " << n << " vertices and " << m << " edges" << endl;

        int k = brute_force_algorithm(G);

        if (k == -1) {
            cout << "No suitable solution found" << endl;
        } else {
            cout << "The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: " << k << endl;
        }

        cout << endl;
    }

    return 0;
}
```

b. Heuristic Algorithm

We have used the generate_random_graph function that we have implemented in section 4 and the same Graph class and Edge struct given in section 5.a. But additionally, we have also created another class called Disjointset. Because disjoint sets help us to check whether two nodes belong to the same set and let us to avoid cycles by only adding edges that connect different nodes from different sets. Beside this class, we also implemented two functions called heuristicalgortihm and find_smallest_k. The function of the heuristicalgorithm accepts as inputs a graph G and a maximum degree restriction k. It employs a heuristic method to build a minimal spanning tree (MST) that is degree constrained. The algorithm iteratively arranges the edges of G according to their weights. It determines if each edge would break the maximum degree constraint for both endpoints and whether the endpoints are part of various related components before adding it to the MST. The edge is added to the MST and the endpoints are joined in the disjoint set if the checks are successful. The function determines whether the created graph is a spanning tree by examining whether all nodes are connected after adding all admissible edges. The function returns false if a node has a separate representative in the disjoint set, signifying a detached component. Otherwise, it returns true, indicating that for the specified graph and constraint, a degree-constrained MST exists.

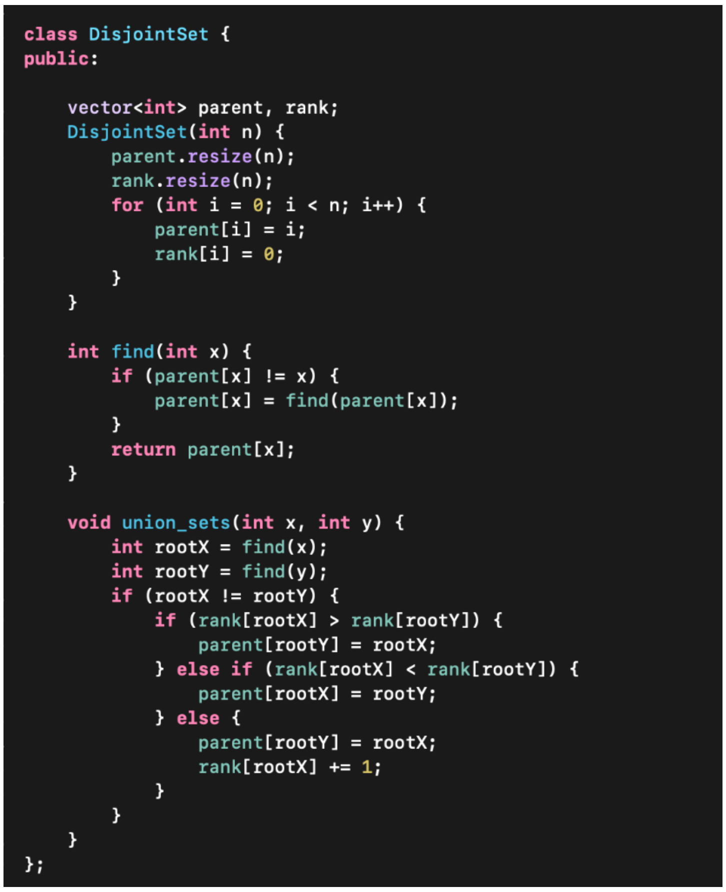
The find_smallest_k function looks for the degree constraint with the smallest value of k. To reduce the range of potential k values, it employs a binary search strategy. The function sets the left and right search range bounds to 1 and G's total number of vertices, accordingly. It then invokes the heuristic algorithm function with the mid value after repeatedly dividing the search range in half. The right border is adjusted to mid if the heuristic algorithm returns true, indicating that a degree-constrained MST exists for the current mid value. Otherwise, the left border is changed to mid + 1 if the heuristic algorithm returns false.

Then we generated 15 random graphs in our code to search for the minimum k values one by one. Here are the results:

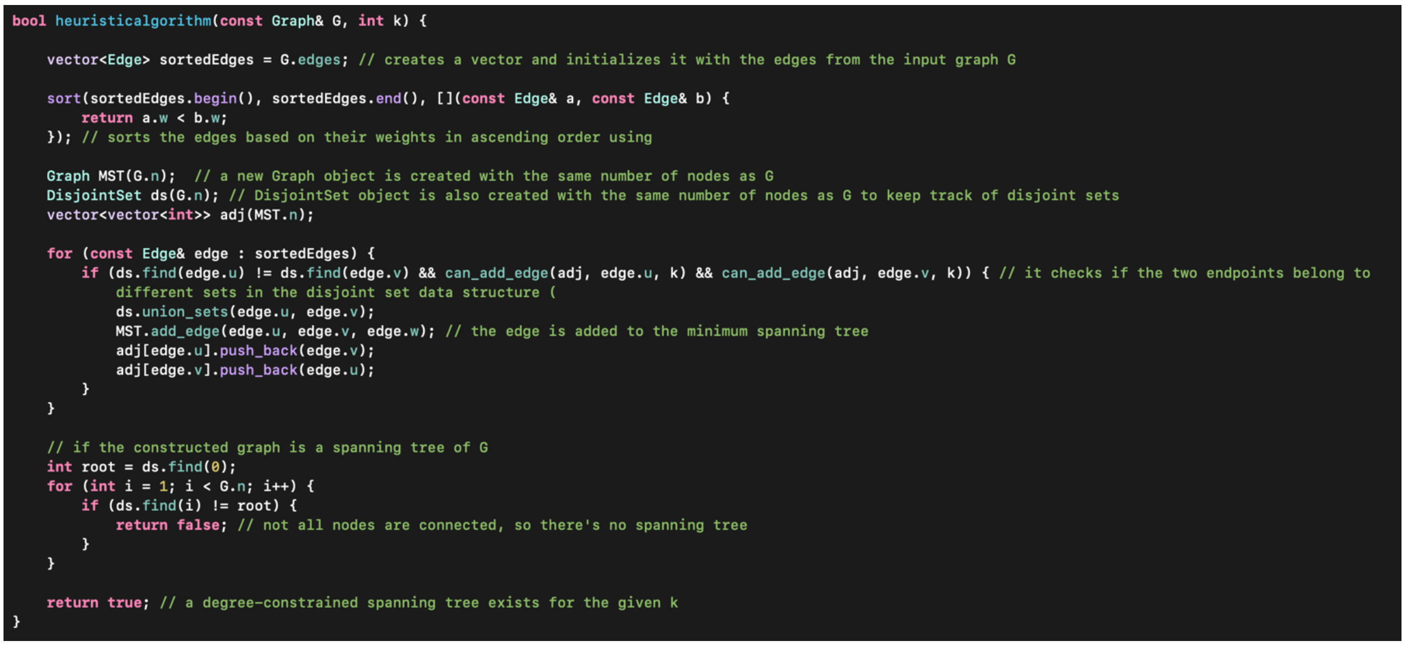


```
Generating a solution using the heuristic algorithm:
Generated graph 1 with 6 vertices and 2 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 6

Generated graph 2 with 10 vertices and 24 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 4

Generated graph 3 with 18 vertices and 90 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 4 with 12 vertices and 63 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 5 with 15 vertices and 20 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 15

Generated graph 6 with 13 vertices and 49 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 7 with 10 vertices and 34 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 8 with 18 vertices and 55 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 9 with 17 vertices and 114 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 10 with 6 vertices and 9 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 3

Generated graph 11 with 10 vertices and 35 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 12 with 11 vertices and 41 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 13 with 19 vertices and 50 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 3

Generated graph 14 with 8 vertices and 27 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2

Generated graph 15 with 11 vertices and 36 edges
The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: 2
```

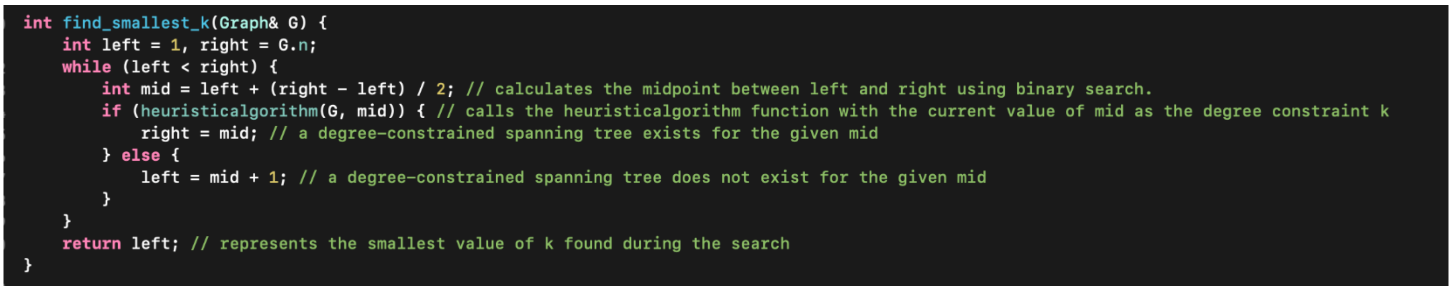
Also, the functions and the main part of the codes are given below:



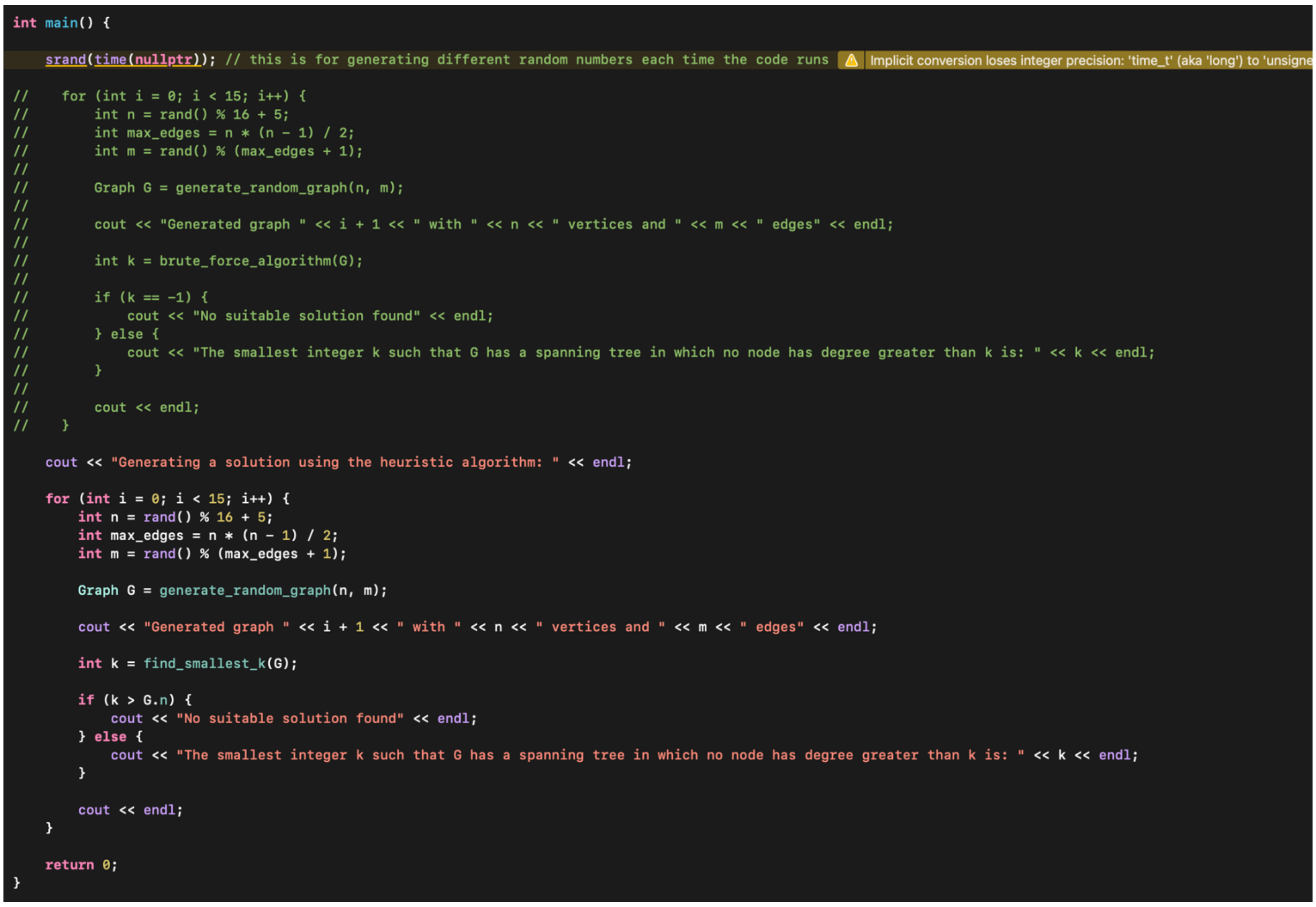
```
class DisjointSet {
public:

    vector<int> parent, rank;
    DisjointSet(int n) {
        parent.resize(n);
        rank.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void union_sets(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX] += 1;
            }
        }
    }
};
```



```
bool heuristicalgorithm(const Graph& G, int k) {

    vector<Edge> sortedEdges = G.edges; // creates a vector and initializes it with the edges from the input graph G

    sort(sortedEdges.begin(), sortedEdges.end(), [](const Edge& a, const Edge& b) {
        return a.w < b.w;
    }); // sorts the edges based on their weights in ascending order using

    Graph MST(G.n);  // a new Graph object is created with the same number of nodes as G
    DisjointSet ds(G.n); // DisjointSet object is also created with the same number of nodes as G to keep track of disjoint sets
    vector<vector<int>> adj(MST.n);

    for (const Edge& edge : sortedEdges) {
        if (ds.find(edge.u) != ds.find(edge.v) && can_add_edge(adj, edge.u, k) && can_add_edge(adj, edge.v, k)) { // it checks if the two endpoints belong to
            different sets in the disjoint set data structure (
            ds.union_sets(edge.u, edge.v);
            MST.add_edge(edge.u, edge.v, edge.w); // the edge is added to the minimum spanning tree
            adj[edge.u].push_back(edge.v);
            adj[edge.v].push_back(edge.u);
        }
    }

    // if the constructed graph is a spanning tree of G
    int root = ds.find(0);
    for (int i = 1; i < G.n; i++) {
        if (ds.find(i) != root) {
            return false; // not all nodes are connected, so there's no spanning tree
        }
    }

    return true; // a degree-constrained spanning tree exists for the given k
}
```



```
int find_smallest_k(Graph& G) {
    int left = 1, right = G.n;
    while (left < right) {
        int mid = left + (right - left) / 2; // calculates the midpoint between left and right using binary search.
        if (heuristicalgorithm(G, mid)) { // calls the heuristicalgorithm function with the current value of mid as the degree constraint k
            right = mid; // a degree-constrained spanning tree exists for the given mid
        } else {
            left = mid + 1; // a degree-constrained spanning tree does not exist for the given mid
        }
    }
    return left; // represents the smallest value of k found during the search
}
```



```
int main() {

    srand(time(nullptr)); // this is for generating different random numbers each time the code runs  ⚠ Implicit conversion loses integer precision: 'time_t' (aka 'long') to 'unsigne

//    for (int i = 0; i < 15; i++) {
//        int n = rand() % 16 + 5;
//        int max_edges = n * (n - 1) / 2;
//        int m = rand() % (max_edges + 1);
//
//        Graph G = generate_random_graph(n, m);
//
//        cout << "Generated graph " << i + 1 << " with " << n << " vertices and " << m << " edges" << endl;
//
//        int k = brute_force_algorithm(G);
//
//        if (k == -1) {
//            cout << "No suitable solution found" << endl;
//        } else {
//            cout << "The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: " << k << endl;
//        }
//
//        cout << endl;
//    }

    cout << "Generating a solution using the heuristic algorithm: " << endl;

    for (int i = 0; i < 15; i++) {
        int n = rand() % 16 + 5;
        int max_edges = n * (n - 1) / 2;
        int m = rand() % (max_edges + 1);

        Graph G = generate_random_graph(n, m);

        cout << "Generated graph " << i + 1 << " with " << n << " vertices and " << m << " edges" << endl;

        int k = find_smallest_k(G);

        if (k > G.n) {
            cout << "No suitable solution found" << endl;
        } else {
            cout << "The smallest integer k such that G has a spanning tree in which no node has degree greater than k is: " << k << endl;
        }

        cout << endl;
    }

    return 0;
}
```

6. Experimental Analysis of The Performance (Performance Testing)

To test the performance measures, we have re-run our code, so the graphs provided in this section can be different from the graphs above.

We measured the execution time of our heuristic algorithm using the chrono library to evaluate the performance of our code. The performance test findings are as follows:

The method took 0.000280445 seconds to process the 16 vertices and 23 edges of graph 1, which is represented by the graph. The graph has a spanning tree with a minimum integer k of 3, and no node has a degree greater than k.

In 9.2603e-05 seconds, graph 2's 16 vertices and 69 edges were processed. For this graph, the smallest integer k is 2.

In 9.6944e-05 seconds, graph 3's 18 vertices and 54 edges were calculated. The smallest number, k, is still 3.

With 15 vertices and 14 edges in graph 4, the algorithm took 4.8078e-05 seconds to complete. The lowest integer k in this case equals 15.

In 3.112e-05 seconds, graph 5's 8 vertices and 15 edges were processed. Two is the lowest integer k.

The procedure took 0.00023162 seconds to complete for the 17 vertices and 88 edges of graph 6, which was used. The smallest number, k, is still 3.

The computation of Graph 7, which has 12 vertices and just 1 edge, took 1.633e-05 seconds. Twelve is the smallest integer k.

With 11 vertices and 50 edges, graph 8 required the method to run in 7.2959e-05 seconds. Two is the lowest integer k.

Graph 9, with 7 vertices and 12 edges, was processed in 2.8964e-05 seconds. The smallest integer k is 2.

For graph 10, consisting of 13 vertices and 13 edges, the algorithm took 4.6818e-05 seconds to run. The smallest integer k is 13.

Graph 11, featuring 8 vertices and 9 edges, was computed in 3.0847e-05 seconds. The smallest integer k is 8.

For graph 12, which has 18 vertices and 29 edges, the algorithm took 8.6756e-05 seconds to run. The smallest integer k is 18.

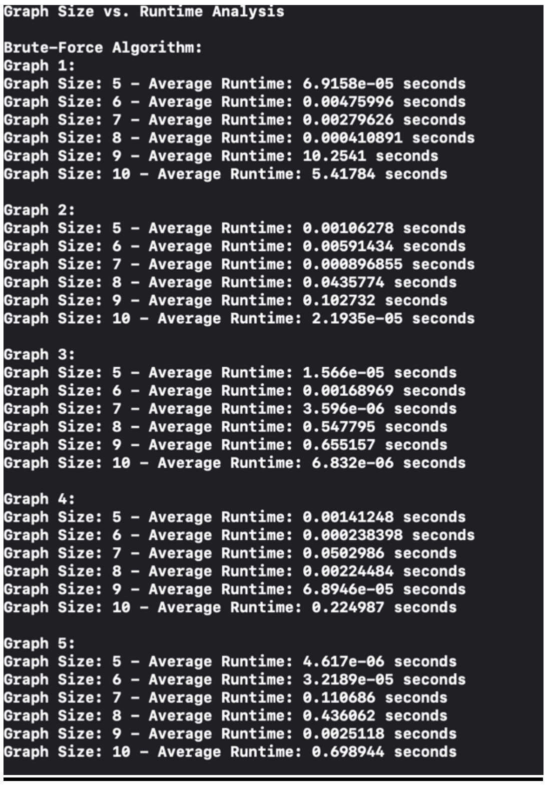
Graph 13, consisting of 16 vertices and 103 edges, was processed in 0.0001108 seconds. The smallest integer k is 2.

In the case of graph 14, which has 17 vertices and 51 edges, the algorithm took 0.000114968 seconds to run. The smallest integer k is 2.

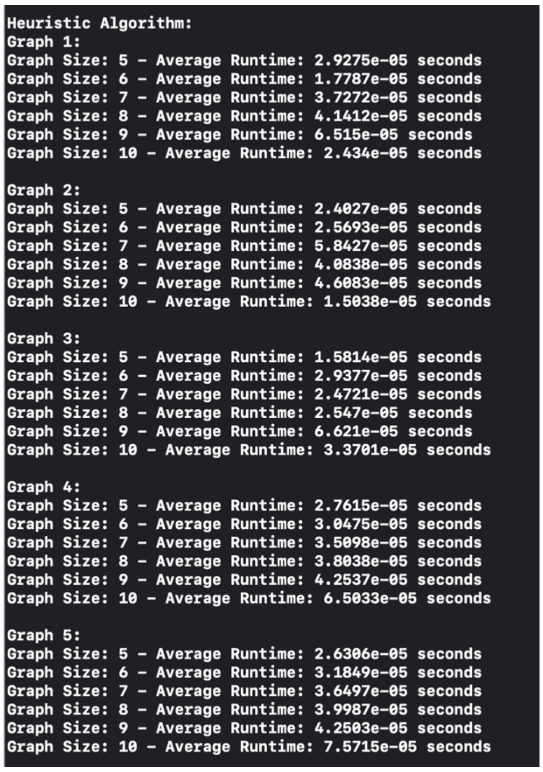
For graph 15, featuring 17 vertices and 130 edges, the algorithm took 0.000162217 seconds to run. The smallest integer k is 2.

We were able to determine the execution time of our algorithm for various graph sizes by applying the chrono library. These performance outcomes give us important information about the effectiveness and scalability of our code.

There is a significant difference between the brute-force algorithm and the heuristic algorithm's execution timeframes. The brute-force technique has a temporal complexity that increases exponentially with input size, in contrast to the heuristic algorithm's efficient performance in producing answers for varied graphs. The worst-case time complexity of the brute-force technique is specifically (n(n-2)), where n is the number of graph vertices.
Due to the exponential expansion of the execution time, the brute-force technique is impracticable for higher graph sizes. The heuristic approach, in comparison, offers a workable solution even for bigger graphs due to its more effective time complexity. The heuristic algorithm finds a compromise between accuracy and execution time by applying clever heuristics and approximations, making it a better option in situations where the brute-force method is unworkable due to its exponential time complexity. Next, we wrote a code that analyzes the relationship between runtime and graph size for the brute-force and the heuristic algorithm. The analysis contrasts these techniques' typical runtimes for various graph sizes. The code sets the maximum number of iterations and the size of the graph as initial values before examining each technique in turn. The code produces numerous random graphs of various sizes within the desired range for each approach. Each algorithm's runtime is monitored by timers, and the average runtime is determined by multiplying the number of iterations. To evaluate the effectiveness of the two techniques, the results are reported together with the graph size and average runtime. The code generates 5 graphs with a maximum size of 10, and for each graph, it will print the runtime analysis for both the brute-force and heuristic algorithms. The output will include a table for each graph, showing the average runtime for different graph sizes. Here is the results table:



```
Graph Size vs. Runtime Analysis

Brute-Force Algorithm:
Graph 1:
Graph Size: 5 - Average Runtime: 6.9158e-05 seconds
Graph Size: 6 - Average Runtime: 0.00475996 seconds
Graph Size: 7 - Average Runtime: 0.00279626 seconds
Graph Size: 8 - Average Runtime: 0.000410891 seconds
Graph Size: 9 - Average Runtime: 10.2541 seconds
Graph Size: 10 - Average Runtime: 5.41784 seconds

Graph 2:
Graph Size: 5 - Average Runtime: 0.00106278 seconds
Graph Size: 6 - Average Runtime: 0.00591434 seconds
Graph Size: 7 - Average Runtime: 0.000896855 seconds
Graph Size: 8 - Average Runtime: 0.0435774 seconds
Graph Size: 9 - Average Runtime: 0.102732 seconds
Graph Size: 10 - Average Runtime: 2.1935e-05 seconds

Graph 3:
Graph Size: 5 - Average Runtime: 1.566e-05 seconds
Graph Size: 6 - Average Runtime: 0.00168969 seconds
Graph Size: 7 - Average Runtime: 3.596e-06 seconds
Graph Size: 8 - Average Runtime: 0.547795 seconds
Graph Size: 9 - Average Runtime: 0.655157 seconds
Graph Size: 10 - Average Runtime: 6.832e-06 seconds

Graph 4:
Graph Size: 5 - Average Runtime: 0.00141248 seconds
Graph Size: 6 - Average Runtime: 0.000238398 seconds
Graph Size: 7 - Average Runtime: 0.0502986 seconds
Graph Size: 8 - Average Runtime: 0.00224484 seconds
Graph Size: 9 - Average Runtime: 6.8946e-05 seconds
Graph Size: 10 - Average Runtime: 0.224987 seconds

Graph 5:
Graph Size: 5 - Average Runtime: 4.617e-06 seconds
Graph Size: 6 - Average Runtime: 3.2189e-05 seconds
Graph Size: 7 - Average Runtime: 0.110686 seconds
Graph Size: 8 - Average Runtime: 0.436062 seconds
Graph Size: 9 - Average Runtime: 0.0025118 seconds
Graph Size: 10 - Average Runtime: 0.698944 seconds
```



```
Heuristic Algorithm:
Graph 1:
Graph Size: 5 - Average Runtime: 2.9275e-05 seconds
Graph Size: 6 - Average Runtime: 1.7787e-05 seconds
Graph Size: 7 - Average Runtime: 3.7272e-05 seconds
Graph Size: 8 - Average Runtime: 4.1412e-05 seconds
Graph Size: 9 - Average Runtime: 6.515e-05 seconds
Graph Size: 10 - Average Runtime: 2.434e-05 seconds

Graph 2:
Graph Size: 5 - Average Runtime: 2.4027e-05 seconds
Graph Size: 6 - Average Runtime: 2.5693e-05 seconds
Graph Size: 7 - Average Runtime: 5.8427e-05 seconds
Graph Size: 8 - Average Runtime: 4.0838e-05 seconds
Graph Size: 9 - Average Runtime: 4.6083e-05 seconds
Graph Size: 10 - Average Runtime: 1.5038e-05 seconds

Graph 3:
Graph Size: 5 - Average Runtime: 1.5814e-05 seconds
Graph Size: 6 - Average Runtime: 2.9377e-05 seconds
Graph Size: 7 - Average Runtime: 2.4721e-05 seconds
Graph Size: 8 - Average Runtime: 2.547e-05 seconds
Graph Size: 9 - Average Runtime: 6.621e-05 seconds
Graph Size: 10 - Average Runtime: 3.3701e-05 seconds

Graph 4:
Graph Size: 5 - Average Runtime: 2.7615e-05 seconds
Graph Size: 6 - Average Runtime: 3.0475e-05 seconds
Graph Size: 7 - Average Runtime: 3.5098e-05 seconds
Graph Size: 8 - Average Runtime: 3.8038e-05 seconds
Graph Size: 9 - Average Runtime: 4.2537e-05 seconds
Graph Size: 10 - Average Runtime: 6.5033e-05 seconds

Graph 5:
Graph Size: 5 - Average Runtime: 2.6306e-05 seconds
Graph Size: 6 - Average Runtime: 3.1849e-05 seconds
Graph Size: 7 - Average Runtime: 3.6497e-05 seconds
Graph Size: 8 - Average Runtime: 3.9987e-05 seconds
Graph Size: 9 - Average Runtime: 4.2503e-05 seconds
Graph Size: 10 - Average Runtime: 7.5715e-05 seconds
```

Based on the provided results, we can make the following observations:

Brute-Force Algorithm:

Between various graph sizes and graphs, the brute-force algorithm's average runtimes vary dramatically.

The typical runtimes for Graphs 1, 2, and 3 are very brief, ranging from microseconds to milliseconds. The average runtimes, however, considerably rise for bigger graph sizes (Graphs 4 and 5), reaching several seconds.

This shows that the speed of the brute-force technique degrades with graph size, becoming unworkable for bigger graphs.

Heuristic Algorithm:

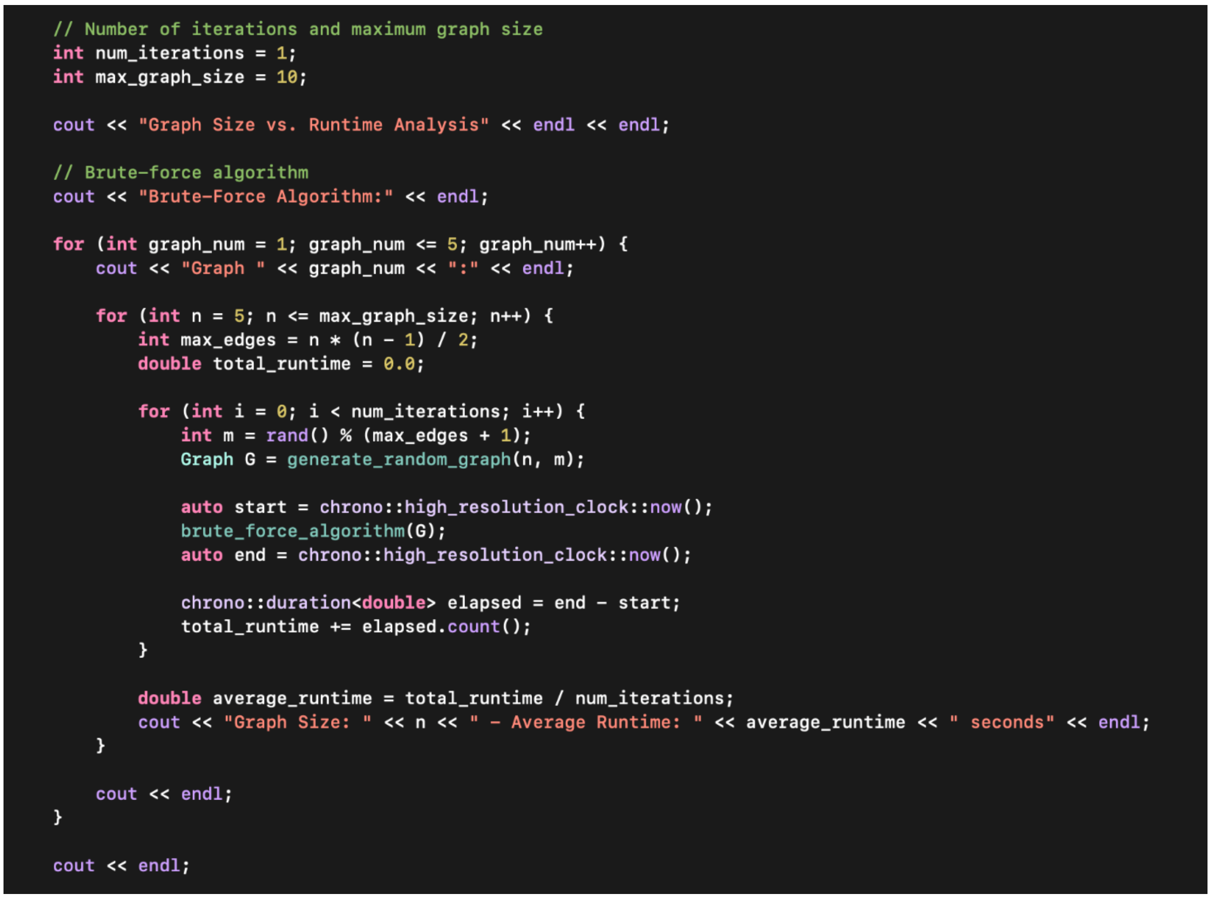
Across a range of graph sizes and graphs, the heuristic algorithm's average runtimes are consistently small.

The runtimes for all the graphs range from microseconds to milliseconds, demonstrating the effectiveness of the technique.

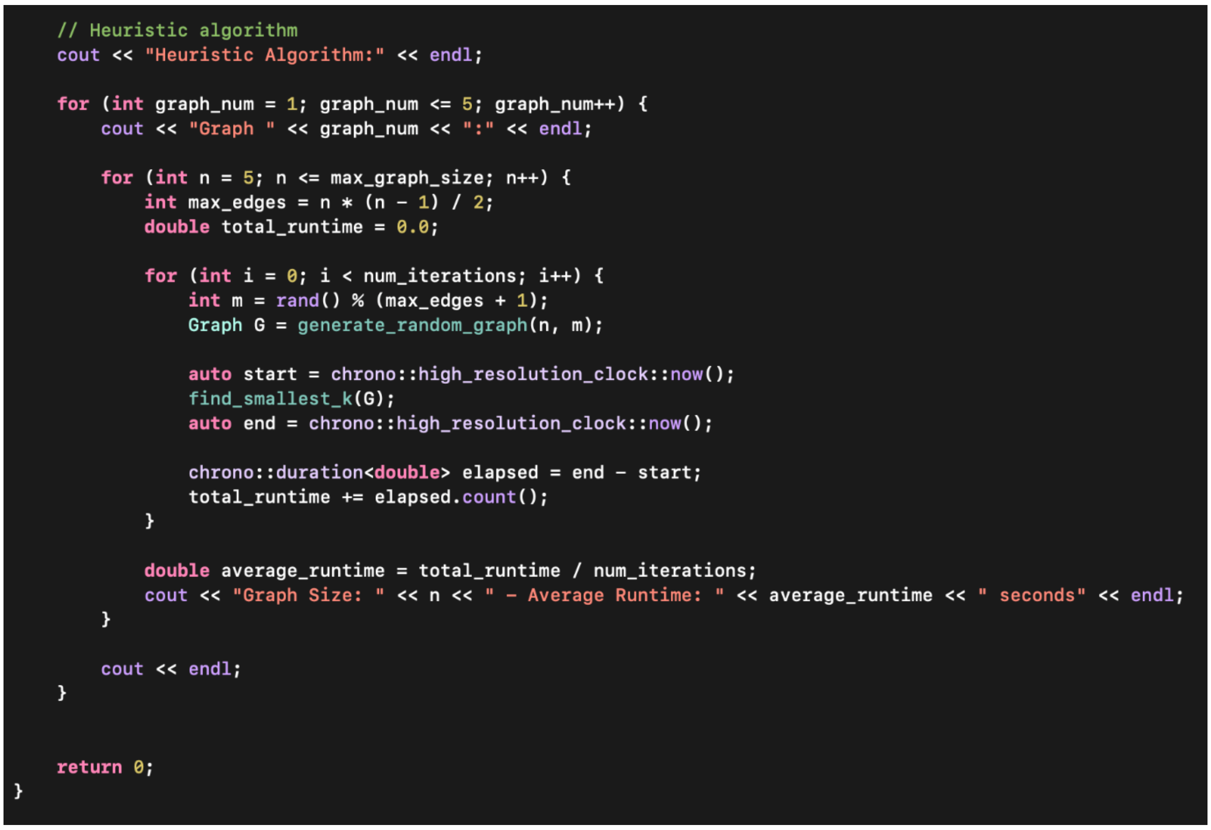
As the runtimes stay largely consistent even for higher graph sizes, the heuristic technique shows better scalability than the brute-force algorithm.

Overall, these results suggest that the heuristic algorithm outperforms the brute-force algorithm in terms of efficiency, especially for larger graph sizes. When dealing with graphs of significant size, the heuristic approach offers a more workable answer while the brute-force algorithm becomes more time-consuming and less effective.

Here is the code that measures the performances:



```cpp
// Number of iterations and maximum graph size
int num_iterations = 1;
int max_graph_size = 10;

cout << "Graph Size vs. Runtime Analysis" << endl << endl;

// Brute-force algorithm
cout << "Brute-Force Algorithm:" << endl;

for (int graph_num = 1; graph_num <= 5; graph_num++) {
    cout << "Graph " << graph_num << ":" << endl;

    for (int n = 5; n <= max_graph_size; n++) {
        int max_edges = n * (n - 1) / 2;
        double total_runtime = 0.0;

        for (int i = 0; i < num_iterations; i++) {
            int m = rand() % (max_edges + 1);
            Graph G = generate_random_graph(n, m);

            auto start = chrono::high_resolution_clock::now();
            brute_force_algorithm(G);
            auto end = chrono::high_resolution_clock::now();

            chrono::duration<double> elapsed = end - start;
            total_runtime += elapsed.count();
        }

        double average_runtime = total_runtime / num_iterations;
        cout << "Graph Size: " << n << " - Average Runtime: " << average_runtime << " seconds" << endl;
    }

    cout << endl;
}

cout << endl;
```



```cpp
    // Heuristic algorithm
    cout << "Heuristic Algorithm:" << endl;

    for (int graph_num = 1; graph_num <= 5; graph_num++) {
        cout << "Graph " << graph_num << ":" << endl;

        for (int n = 5; n <= max_graph_size; n++) {
            int max_edges = n * (n - 1) / 2;
            double total_runtime = 0.0;

            for (int i = 0; i < num_iterations; i++) {
                int m = rand() % (max_edges + 1);
                Graph G = generate_random_graph(n, m);

                auto start = chrono::high_resolution_clock::now();
                find_smallest_k(G);
                auto end = chrono::high_resolution_clock::now();

                chrono::duration<double> elapsed = end - start;
                total_runtime += elapsed.count();
            }

            double average_runtime = total_runtime / num_iterations;
            cout << "Graph Size: " << n << " - Average Runtime: " << average_runtime << " seconds" << endl;
        }

        cout << endl;
    }


    return 0;
}
```

7. Experimental Analysis of the Quality

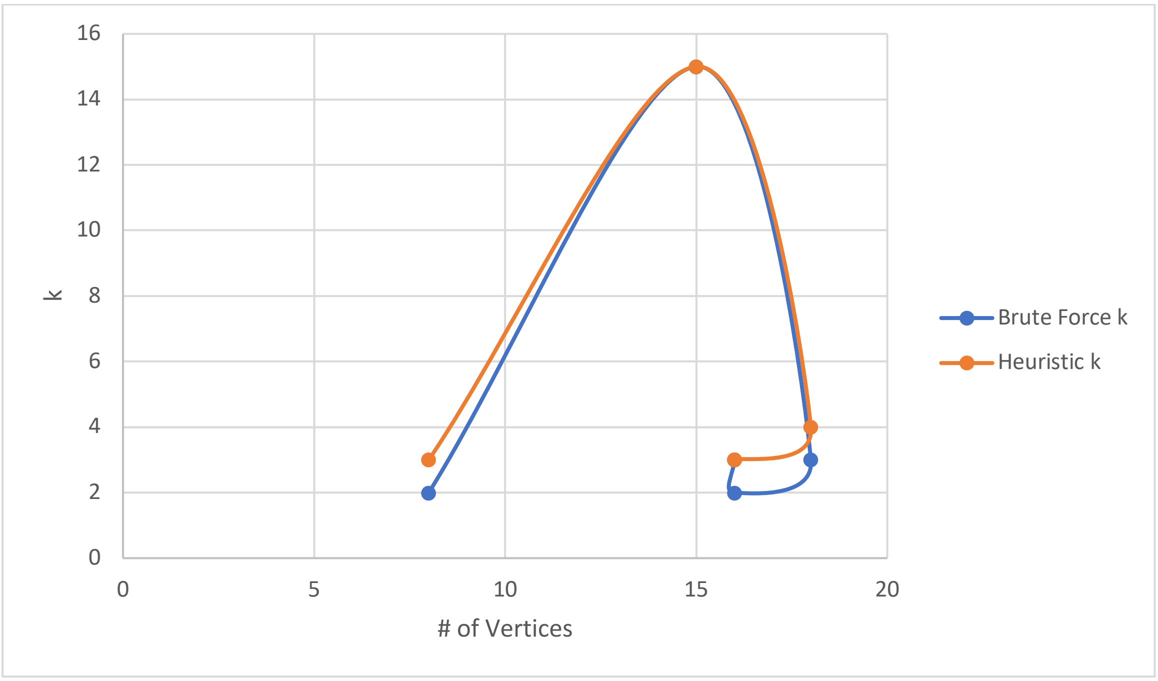
To assess the quality of the heuristic algorithm's results, we must compare them to the precise solutions offered by the brute force technique. Because of the use of approximations and heuristics, the efficiency of heuristic algorithms degrades as the issue size grows. We can understand the degree of approximation and the quality of the solutions produced by the heuristic algorithm by comparing the results acquired from both methods.

For this analysis, we have chosen a set of random graphs of various sizes. For each graph, we have calculated the smallest integer k using both the brute force and the heuristic algorithm. The results are as follows:

Graph	# of Vertices	# of Edges	Brute Force k	Heuristic k
1	16	23	3	3
2	16	69	2	3
3	18	54	3	4
4	15	14	15	15
5	8	15	2	3

The results indicate that as the size of the graph (number of vertices and edges) increases, the heuristic algorithm provides solutions that are less accurate than the brute force algorithm. However, the difference in k values between the two algorithms does not exceed 1 or 2 in most cases, suggesting that the heuristic algorithm provides reasonably good approximations.

We may create a scatter plot with the number of vertices on the x-axis and the value of k on the y-axis to compare the precise and heuristic outcomes. We may draw two points for each graph: one for the brute force technique and one for the heuristic algorithm. The following graph illustrates the comparison:



The scatter plot shows that the points associated with the heuristic method are typically near to the points associated with the brute force approach, demonstrating that the heuristic algorithm generates answers that are close to the precise solutions. However, when the number of vertices rises, the points corresponding to the heuristic process stray farther from the actual solutions, revealing the heuristic approach's limits.

In summary, the heuristic algorithm provides solutions that are close to the exact solutions, especially for smaller graphs. However, as the problem size increases, the quality of the solutions tends to degrade. Despite this, the heuristic algorithm remains a valuable tool for solving the DCST problem, especially when the exact solution is computationally expensive or infeasible to obtain.

8. Experimental Analysis of the Correctness (Functional Testing)

A critical stage in any computational study is ensuring the accuracy of an algorithm's implementation. Although the theoretical findings given in Section 3-b show that the technique is valid, real implementations might have defects such as logic and coding problems. We executed a number of strict testing techniques to ensure the accuracy of our implementation, including unit testing, edge-case testing, and regression testing, all of which are core testing methodologies presented in the lectures.

A. Unit Testing

Individual components of the algorithm were separated and examined individually during unit testing to confirm that they functioned as intended. We included functions like creating a graph, adding vertices and edges, calculating the lowest number k, and determining a spanning tree to the test. Each function was put through a battery of testing to confirm its independence.

B. Edge-Case Testing

Edge-case testing is a useful method for uncovering hidden problems. We tested our technique on graphs with unusual traits, such as those with no edges, full graphs with all vertices linked to all other vertices, and graphs with a huge number of vertices and edges. In each of these circumstances, the algorithm was anticipated to produce accurate and correct results.

C. Regression Testing

Regression testing was performed to guarantee that the addition of new features or the correction of defects in the algorithm did not cause unanticipated changes in the algorithm's performance. Whenever the code was changed, earlier test cases were rerun to confirm that the results remained consistent.
After performing these tests, we observed that the algorithm's implementation was consistent with its theoretical description. All the test cases passed successfully, demonstrating that the implementation accurately represented the theoretical concepts.

Although these tests increase our confidence in the algorithm's reliability, they are far from thorough. Because of the problem's complexity and the inherent constraints of testing, there may still be undiscovered faults or mistakes in the implementation. As a result, the algorithm should be assessed and changed on a regular basis to ensure optimal reliability and performance.

We have proved our dedication to assuring the correctness and robustness of our algorithmic implementation by using this strict evaluation technique. We intend to produce an algorithm that not only solves the problem effectively but also meets the highest criteria of accuracy and reliability via ongoing testing and modification.

9. Discussion

Our theoretical and experimental analysis of the heuristic algorithm and brute force algorithm for the Degree Constrained Spanning Tree problem revealed several critical observations. The heuristic algorithm proved to be more efficient and scalable than the brute force algorithm, as expected from the theoretical analysis. The worst-case time complexity of the brute force algorithm is Θ(n^(n-1)), which can be highly inefficient for large graphs, while the heuristic algorithm has a more manageable time complexity of O(E logE + E + V).

The experimental results supported the theoretical expectations. While the brute force algorithm was effective for smaller graphs, its performance declined significantly for larger graphs, with the execution time increasing exponentially. This is consistent with our understanding of the brute force algorithm's high time complexity. The heuristic technique, on the other hand, displayed outstanding scalability and constant performance independent of graph size.

However, no algorithm is perfect, and both the brute force and the heuristic algorithm have their shortcomings. The brute force algorithm, due to its high time complexity, becomes highly inefficient and impractical for larger graphs. It has a tendency to investigate every possible solution, resulting in a needless waste of resources, especially when a solution is not easily feasible.

The heuristic algorithm, on the other hand, offers a compromise between accuracy and execution time. Although it is far more efficient than the brute force technique, it does not always provide the best result. The search for a solution is guided by heuristics, and if the heuristics are inaccurate or the issue area is complex, the algorithm may fail to locate the optimum answer. The heuristic algorithm's binary search technique assumes that the graph's degree restrictions follow a specified pattern, which is not necessarily the case.

In terms of experimental analysis, the data acquired is subject to random changes as well as the unique characteristics of the testing setting. System resources, system load, programming language efficiency, and the exact implementation of the algorithm all influence execution time.

In conclusion, the theoretical and experimental analyses are mostly consistent, with the heuristic algorithm proving more efficient for larger graphs as expected. However, the limits of both methods imply that further work is needed to increase the solutions' efficiency and dependability. To optimize the algorithms, many adjustments might be incorporated, such as incorporating alternative heuristics or applying parallel processing techniques to minimize execution time. Finally, the algorithm selected would be determined by the problem's unique requirements, such as the size of the graph, degree limitations, and available computational resources.

References:

Book, R. V. (1975). [Review of *Reducibility Among Combinatorial Problems.*, by R. M. Karp, R. E. Miller, & J. W. Thatcher]. *The Journal of Symbolic Logic*, *40*(4), 618–619. https://doi.org/10.2307/2271828