ASSIGNMENT 1

Algorithms & Complexity

VIKENSA GRABOCKA

Table of Contents

Introduction

This project aims to solve the problem of finding the cheapest cost of setting up a computing grid used to predict weather. The weather forecast is conducted using powerful machines named dynos, which make use of sensor data named buckets. Each dyno should either host the bucket itself or get it from another dyno. The number of dynos, the number of possible bonds, the bucket cost, the bond cost and the pairs of dynos that can be connected are received from an input file, whose path is provided by the user. Afterwards the program computes the total cheapest cost, the number of different cheapest solutions and it visualizes one cheapest solution.

Implementation

This project is implemented using Java. It consists of 3 classes. The first class is named BigWeatherConfiguration. It consists of methods and attributes that are needed to evaluate the cost of the cheapest solution and the number of cheapest solutions. The second class is named Visualizer, and it contains methods that are used to visualize one of the cheapest solutions. There is another class called Main which is the starting class of the program.

Data Structures

The Weather Forecasting problem can be translated into a graph problem. Basically, the dynos represent the nodes of a graph, while the bonds represent the edges of the graph.

The data structures used in this program are:

1. ArrayList<ArrayList<Integer>>> which is used to store the adjacency list of the graph. Using an adjacency list to store the information related to the graph has some advantages:
    - In terms of space complexity, it allocates a fixed space to represent the nodes (n, the number of nodes) and some variable space to represent the edges. Since there is a bidirectional communication between the dynos, it means that the graph under consideration is an undirected graph, therefore the amount of space needed to represent the edges is twice the number of edges (2e). So, the space complexity of the adjacency list is:

      $$n + 2e = O(n + e)$$

      The adjacency matrix on the other hand uses a two-dimensional array to represent the graph. Given that u and v are two nodes of the graph the entry

(u,v) = 1 if there exists an edge from u to v otherwise it is 0. Therefore, no matter how dense the graph is, the space complexity of the adjacency matrix is:

$$O(n^2)$$

, where n represents the number of nodes. But the number of edges in a graph is usually << n^2. Therefore,

$$O(n+e) \ll O(n^2)$$

which results in a wasted amount of space in case of an adjacency matrix.

- In terms of time complexity, accessing one specific node in the adjacency list implemented using Array List is constant. Array List is index-based therefore accessing one node has

  $$O(1)$$

  time complexity. But checking if one edge exists might result in linear time complexity, since all the edges that are associated with a particular node should be scanned. In contrast, the adjacency matrix implements this behavior in constant time. Nevertheless, this behavior is not needed in the implementation of my program. The behavior that my algorithms use is the one that gets all the edges associated with a particular node. In this case the adjacency list and the adjacency matrix provide the same time complexity:

  $$O(1)$$

  to get the desired node, while to get the desired edge an amount of time proportional to the degree of the desired node is needed, which in the worst case might be equal to the number of edges. Therefore, using an adjacency list or an adjacency matrix does not make a difference for this use case.

  The adjacency list might have been implemented using a Linked List instead of an Array List. But this implementation would have led to a

  $$O(n)$$

  time complexity even for identifying a particular node, since the Linked List implementation requires the traversal over all the elements to find a specific element.

2. boolean []
   An array of boolean values named visited is used to keep track of visited nodes during the DFS. It ensures constant time complexity for reading and writing operations. Getting or modifying the corresponding boolean value for a specific node has
   $$O(1)$$
   time complexity since the array is index based.

3. Set<List<Integer>>, more concretely HashSet
   A Set<List<Integer>> named edgesPerComponent is used to keep track of the edges for a connected component. Basically, this set is used as an input for the method that builds the Laplacian matrix that is used to count the number of spanning trees. In order not to store repeated edges the best data structure is a Set. This set consists of lists of integers which represent the edges. List of Integer is used instead of arrays, since in the array based approach although two arrays might contain the same values they will not be detected as equal and will be inserted into the set, because for the array based implementation they will be treated as equal only if they are the same object in memory, while the list based implementation will detect equality based on value.

4. long [] []
   A two-dimensional array of long data type is used to build the Laplacian matrix, which is used to count the number of spanning trees. The two-dimensional array is more suitable for performing matrix operations.

5. int []
   An array of int values (int []) named prev is used to keep track of the nodes in the tree structure that is formed for each connected component. Basically, the index of the array represents each node and for each node is recorded the node immediately before it on the path. The array implementation allows O (1) time complexity for performing read and write operations.

Algorithms

Pseudocode

Algorithm 1 DFS(G)

Input: G - adjacency list of the graph
Output: connectedAndCombinations array - an array that stores the number of connected components and the number of combinations

```
1: function DFS(G)
2: connectedAndCombinations ← []
3: numberOfCombinations ← 1
4: for each node v ∈ V do
5:    visited[v] ← false
6: end for
7: for each node v ∈ V do
8:    nodesPerComponent ← 1
9:    if not visited(v) then
10:      prev[i] ← 0
11:      EXPLORE(G, u)
12:      if bucketCost == bondCost then
13:         numberOfCombinations ← nodesPerComponent × numberOfCombinations×countSpanningTrees(nodesPerComponent, edgesPerComponent)× 2^(nodesPerComponent−1)
14:      else if bucketCost < bondCost then
15:         numberOfCombinations ← 1
16:      else
17:         numberOfCombinations ← nodesPerComponent × numberOfCombinations×countSpanningTrees(nodesPerComponent, edgesPerComponent)
18:      end if
19:      ccNum ← ccNum + 1
20:      edgesPerComponent ← ∅
21:   end if
22: end for
23: connectedAndCombinations[0] ← ccNum - 1
24: connectedAndCombinations[1] ← numberOfCombinations
25: return connectedAndCombinations
26: end function
```

Algorithm 2 EXPLORE(G, v)

```
1: function EXPLORE(G, v)
2: visited(v) ← true
3: for each edge (v, u) ∈ E do
4:    pair ← [v,u]
5:    insert(edgesPerComponet, pair)
6:    if not visited(u) then
7:       prev[u] ← v
8:       nodesPerComponnet(v) ← nodesPerComponent + 1
9:       EXPLORE(G, u)
10:   end if
11: end for
12: end function
```

Algorithm 3 countSpanningTrees(*size*, *edges*)

```
function countSpanningTrees(size, edges)
laplacian ← [][]
for each edge (v, u) ∈ E do
    u ← (edges.get(0)-1)mod size
    v ← (edges.get(1)-1) mod size
    laplacian[u][u] ← laplacian[u][u]+1
    laplacian[v][v] ← laplacian[v][v]+1
    laplacian[u][v] ← laplacian[u][v]-1
    laplacian[u][v] ← laplacian[v][u]-1
end for
return determinant(removeRowCol(laplacian, size -1), size -1)
end function
```

Algorithm 4 removeRowCol(*matrix*, *n*)

```
function removeRowCol(matrix, n)
result ← [][]
for i = 0...i = n do
    for j = 0...j = n do
        result[i][j] ← matrix[i][j]
    end for
end for
return result
end function
```

Algorithm 5 determinant (matrix, n)

```
function DETERMINANT(mat, n)
    mod ← 1,000,000,007
    det ← 1
    for i ← 0 to n − 1 do
        pivot ← i
        while pivot < n and mat[pivot][i] = 0 do
            pivot ← pivot + 1
        end while
        if pivot = n then
            return 0
        end if
        if i ≠ pivot then
            Swap mat[i] and mat[pivot]
            det ← −det
        end if
        det ← (det · mat[i][i]) mod mod
        inv ← MODINVERSE(mat[i][i], mod)
        for j ← i + 1 to n − 1 do
            factor ← (mat[j][i] · inv) mod mod
            for k ← i to n − 1 do
                mat[j][k] ← (mat[j][k] − factor · mat[i][k]) mod mod
                if mat[j][k] < 0 then
                    mat[j][k] ← mat[j][k] + mod
                end if
            end for
        end for
    end for
    return (det + mod) mod mod
end function
```

Algorithm 7 computeStatistics()

```
1: function computeStatistics()
2: if bucketCost < bondCost then
3:    totalCost ← bucketCost · dynoNumber
4:    numCheapestSolutions ← 1
5: else
6:    componentsCombinations ← DFS()
7:    numCheapestSolutions ← componentsCombinations[1]
8:    components ← numCheapestSolutions[0]
9:    totalCost ← bucketCost · components + (dynoNumber − components) ·
      bondCost
10: end if
11: display total cost
12: display number of cheapest solutions
13: visualize one solution
14: end function
```

Proof of correctness

1.Calculate total cost

Finding the cheapest solution for the computing grid required to forecast weather becomes a graph problem. Dynos represent nodes, while bonds represent edges, since the communication of dynos is bidirectional, the graph that is used to represent them is an undirected graph. The fact that certain dynos can be connected to each other means that the graph might consist of several connected components. Therefore, it is required to find the cheapest cost for each component.

The cheapest solution for each component is either:

1. The one that consists of all nodes being independent of each other and hosting themselves the bucket.
2. The one that consists of all nodes being connected and only one node hosting the component.
3. The one that consists of k nodes being independent and hosting themselves the bucket and n-k nodes being connected, while only one of them hosting the bucket. $(1 \leq k \leq n-2)$
4. The one that separates the component into k components that have a tree structure, such that $2 \leq k < n$.

Let's denote the cost of the bucket by x and the cost of the bond by y. Let's denote by n the number of nodes in the component. The cost of the component in each case is:

1. $cost = x\ n$
2. $cost = x + y\ (n-1)$
   If all the nodes of a connected component are connected, the minimum number of edges required to connect them is n-1 (It is a spanning tree).
3. $cost = (k+1)\,x + y(n-k-1)$

4. $cost = kx + y(n_1 - 1 + n_2 - 1 + n_3 - 1 + \cdots . + n_k - 1)$ where $n_1, n_2 \ldots . n_k$ represents the number of nodes in each subcomponent.

If $x \geq y$ (cost of one bucket is greater than or equal to the cost of one bond):

-Comparing the cost in the second scenario with the cost in the first scenario:

$$xn \geq x + y(n-1)$$

$$xn - x - y(n-1) \geq 0$$

$$xn - x - yn + y \geq 0$$

$$n(x-y) - (x-y) \geq 0$$

$$(n-1)(x-y) \geq 0$$

which is true for $x \geq y$

since $x - y \geq 0$ and $n - 1 > 0$ for all $n > 1$

Therefore, when the **cost of a bucket is greater than the cost of a bond** connecting all the **dynos together and hosting the bucket in one dyno** is **cheaper** than having all the dynos disconnected and hosting one bucket in each dyno. While when the cost of the bucket is equal to the cost of the bond the cost in both cases is equal.

-Comparing the cost in the second scenario with the cost in the third scenario:

Second scenario: $cost = x + y(n-1) = x + yn - y$

Third scenario: $cost = xk + x + yn - yk - y$

Prove:

$$x + yn - y \leq xk + x + yn - yk - y$$

$$yk - xk \leq 0$$

$$k(y - x) \leq 0$$

which is true if $x \geq y$ and $1 \leq k \leq n - 2$

-Comparing the cost in the second scenario with the cost in the fourth scenario:

Second scenario: $cost = x + y(n-1) = x + yn - y$

Fourth scenario: $cost = kx + y(n_1 - 1 + n_2 - 1 + n_3 - 1 + \cdots . + n_k - 1)$

Prove:

$$x + yn - y \leq kx + y(n_1 - 1 + n_2 - 1 + n_3 - 1 + \cdots . + n_k - 1)$$

$$n_1 + n_2 + \cdots + n_k = n$$

Therefore, the right side of the inequality becomes: $kx + y(n - k)$

$$x + yn - y \leq kx + y(n - k)$$

$$x + yn - y \leq kx + yn - yk$$

$$x + yn - y - kx - yn + yk \leq 0$$

$$x - y - kx + ky \leq 0$$

$$x - y + k(y - x) \leq 0$$

$$-(y - x) + k(y - x) \leq 0$$

$$(y - x)(k - 1) \leq 0$$

which is true if $x \geq y$ and $2 \leq k < n$ since $y - x \leq 0 \; and \; k - 1 > 0$

Therefore, **when the cost of a bond is less than the cost of a bucket the cheapest** solution for each component is achieved in the **second scenario**, when all the nodes are connected and only one of them hosts the bucket. If the cost of a bond is equal to the cost of a bucket the four scenarios provide equal cost.

If x < y

-Comparing the cost in the second scenario with the cost in the first scenario:

First scenario: $cost = x\,n$

Second scenario: $cost = x + y(n - 1)$

Finding out when the minimum cost is achieved:

$$xn < x + y(n - 1)$$

$$xn < x + yn - y$$

$$xn - yn - x + y < 0$$

$$n(x-y)-(x-y)<0$$

$$(n-1)(x-y)<0$$

Which is true, since $x < y \rightarrow x - y < 0$ and $n-1 > 0 \nabla n > 1$

Therefore, when **x < y** the cost in the first scenario is less than the cost in the second scenario.

-Comparing the cost in the first scenario with the cost in the third scenario:

First scenario: $cost = x\,n$

Third scenario: $cost = (k+1)x + y(n-k-1) = xk + x + yn - yk - y$

Finding out when the minimum cost is achieved:

$$x\,n > xk + x + yn - yk - y$$

$$xn - xk - x - yn + yk + y > 0$$

$$n(x-y) + y - x + k(y-x) > 0$$

$$n(x-y) + (k+1)(y-x) > 0$$

$$(k+1) > \frac{n(y-x)}{(y-x)}$$

$$k > n - 1$$

Only for values of k > n -1 the cost in the third scenario is less than the cost in the first scenario.

Since $1 \leq k \leq n-2$, k cannot be greater than n -1. Meaning that the cost in the third scenario cannot be less than the cost in the first scenario when x < y.

-Comparing the cost in the first scenario with the cost in the fourth scenario:

First scenario: $cost = x\,n$

Fourth scenario: $cost = kx + y(n-k)$

$$x\,n < kx + y(n - k)$$

$$xn - kx - yn + yk < 0$$

$$n(x-y)-k(x-y)<0$$

$$(n-k)(x-y)<0$$

This inequality holds since $k < n$ therefore $n - k > 0$ and $x - y < 0$ since $x < y$.

Conclusion

- When the cost of a bucket is greater than the cost of the bond the cheapest solution for each component is the one that consists of all the dynos of the component being connected while only one of them hosting the bucket.
- When the cost of a bucket is less than the cost of a bond the cheapest solution is the one that consists of all the dynos being disconnected and each of them hosting the bucket.
- When the cost of the bucket is equal to the cost of the bond all the above – mentioned scenarios result in equal cost.

For the entire computing grid, the cheapest cost is achieved when the cheapest cost of all the components is summed.

- When the bucket cost is less than the cost of a bond:

  $$total\ cost = xn_1 + xn_2 + \ldots.. + xn_k = x(n_1 + n_2 + \ldots + n_k) = xn$$

  where n_i is the number of nodes (dynos) in each component, n is the total number of nodes (dynos) and k is the number of connected components.
- When the bucket cost is equal to the cost of a bond all the above-mentioned solutions result in equal costs. Therefore, we can choose the solution in which the dynos that belong to a connected component are connected, while only one hosts the bucket. Therefore, the total cost will become:

  $$total\ cost = x + (n_1 - 1)\,y + x + (n_2 - 1)\,y + \ldots.. + x + (n_k - 1)\,y$$
  $$= kx + y(n_1 + n_2 + \ldots + n_k - k) = kx + y(n-k)$$

- When the bucket cost is greater than the bond cost:

  $$total\ cost = x + (n_1 - 1)\,y + x + (n_2 - 1)\,y + \ldots.. + x + (n_k - 1)\,y$$
  $$= kx + y(n_1 + n_2 + \ldots + n_k - k) = kx + y(n-k)$$

  where n_i is the number of nodes (dynos) in each component an n is the total number of nodes (dynos), and k is the number of connected components.

This is performed in the computeStatistics method. The cost of a bond and the cost of a bucket are compared, and the total cost is calculated accordingly for each case.

The algorithm that is used to find the number of connected components is based on the Depth-First Search algorithm. The method that calculates the number of connected components is called DFS. It makes use of the explore method, which computes all the reachable nodes from a particular node. This method only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from v. Let's assume that there is some u that it misses, choose any path from v to u, and look at the last vertex on that path that the procedure visited. Call this node z, and let w be the node immediately after it on the same path. v z w u, so z was visited but w was not. This is a contradiction: while the explore procedure was at node z, it would have noticed w and moved on to it. Therefore, the explore method computes the portion of the graph reachable from a particular point (S. Dasgupta, 2006).

DFS algorithm uses the explore method repeatedly until the entire graph is traversed. It maintains a boolean array which notes which nodes are visited, so that each node is processed only once. After a component is discovered the ccNum (an int variable that stores the number of connected components) is updated so that it can be used for the next component. The number of connected components discovered is equal to the ccNum-1 since after the last component was discovered the ccNum was incremented to be prepared for the next iteration, without knowing that it won't result in the discovery of a new component. The number of connected components (along with the number of combinations that result in the cheapest solution) is stored in an array which is returned by the DFS method.

The computeStatistics method checks first if the cost of the bucket is less than the cost of the bond, afterwards it calculates the price based on the above-mentioned formula for the according scenario. Otherwise, it proceeds with a call to the DFS method. From the array that this method returns, it uses the value in the 0^{th} index which corresponds to the number of connected components and according to the above-mentioned formula the total cost is calculated.

2.Compute the number of cheapest solutions

When the cost of a bucket is less than the cost of a bond the cheapest solution is the one that consists of all the dynos being disconnected and hosting the bucket themselves. Therefore, there is only one possible cheapest solution in this case.

When the cost of a bond is less than the cost of a bucket, the cheapest solution for each connected component (group of dynos that can be connected) is the one that consists of all the nodes of a connected component being connected, while only one of them hosting the bucket. Since the cheapest solution for each component is achieved when all the nodes of a component are connected, finding the total number of cheapest solutions for a component means counting the number of trees that can be constructed from that component and multiplying this by the number of nodes of the component, because the bucket can be hosted by any node of the connected component. Basically, there are two independent events here: the position of the bucket and the position of the bonds. By using the product rule, we can multiply the number of spanning trees (number of possibilities for the position of the bonds) by the number of nodes of the component (number of possibilities for the position of the bucket). The number of cheapest solutions for the entire computing grid is obtained by multiplying the number of cheapest solutions for each component. Again, the way of arrangement in each component is an independent event, therefore we can use the product rule to find the number of possible cheapest solutions for the entire grid.

$$number\ of\ cheapest\ solutions\ = \prod_{i\,=\,0}^{k} n_i\, x_{\mathrm{i}}$$

Where x_{i} is the number of trees for each component and n_i is the number of nodes for each component and k is the number of connected components.

When the cost of a bucket is equal to the cost of a bond, the number of cheapest solutions for each component is equal to the number of all the solutions that do not contain cycles (redundant bonds). Basically, for each tree that can be constructed there are several possibilities, each edge can be present or not, all the trees contain $n_i\ - 1$ edges, where n_i is the number of nodes (dynos) that belong to the component. So, in terms of the bonds arrangement there are 2^{n_i-1} possibilities for each tree of the component. The bucket can be hosted by any node of the component. Therefore, from the product rule we can conclude that for each component there are: $x_{\mathrm{i}}\, n_i\, 2^{n_i-1}$ cheapest solutions: where x_{i} is the number of spanning trees of the component. Again, the way of arrangement in each component is an independent event, therefore we can use the product rule to find the number of possible cheapest solutions for the entire grid.

$$number\ of\ cheapest\ solutions\ = \prod_{i\,=\,0}^{k} n_i\, x_{\mathrm{i}}\, 2^{n_i-1}$$

In the DFS method it is kept track of the number of nodes for each component. This number is reset to 1 (Algorithm 1, line 8) each time a new connected component is discovered, while in the explore method the number of nodes for each component is incremented by 1 each time a new node is discovered (Algorithm 2, line 8). Moreover, in the explore method each pair of edges that can be constructed in the corresponding component is added to a set that consists of lists (pairs of edges). A set is used to avoid duplicate edges and moreover, since the graph is undirected when the edge (u,v) is faced the calculation is performed also for the edge (v,u). Therefore, to avoid double calculations which might lead to incorrect results, the edge (u,v) and (v,u) are represented by the edge (u,v) if u>v or by the edge (v,u) if (u<v). This is achieved by using the comparison in the Explore method before the edge is inserted in the set. After a component is discovered, the set is made empty so that it can be used by the following component (Algorithm 1, line 22).

In the DFS method after a component is fully discovered based on the result of the comparison between the cost of a bond and the cost of the bucket, the number of cheapest solutions is computed according to the above-mentioned scenarios. If the cost of the bucket is less than the cost of a bond the number of cheapest solutions is 1, otherwise another method is called that computes the number of spanning trees for each component. The set of edges and the number of nodes of the component is passed as an argument to the countSpanningTrees methods. The numberOfCombinations variable keeps track of the number of cheapest solutions for the entire computing grid by using the above-mentioned formulas for each scenario. Afterwards the number of solutions is stored in the array connectedAndComponents and this value is displayed by the computeStatistics method.

In order to find the number of trees that can be constructed for each component I have used the Kirchhoff s Matrix-Tree Theorem, which says that the number of spanning trees of a graph is equal to any cofactor of the Laplacian matrix (the determinant of the matrix that results from deleting any row and the corresponding column from Laplacian matrix) (Cvetkovic, Rowlinson, & Simic, 2010). The Laplacian matrix of an undirected unweighted graph G = (V, E) is defined as follows:

$$L(i,j) = \begin{cases} deg(i,j) & if\ \ i = j \\ -1 & if\ i \neq j\ and\ (i,j)\ \in E \\ 0 & otherwise \end{cases}$$

The construction of the Laplacian matrix is performed in algorithm 3. Given a set of lists of pairs of edges and the number of nodes as input, the method builds the Laplacian matrix. Since the Laplacian edge is 0 based and our nodes are 1 based, from the number that represents each node is subtracted 1, the modulus operator is used in order to avoid array index out of bounds. This method will run for each connected component and the value that

represents a node might be greater than the index of the array that is used to represent it. For example, a component might have 4 nodes, each of which represented by the values (5,6,7,8), while the Laplacian matrix is initialized to have a size 4x4 meaning that its indices are from (0,0) up to (3,3).

Each time an edge (u,v) is encountered, the entry (u,u) and (v,v) is incremented in the matrix, meaning that at the end the entry (u,u) and (v,v) will reflect the degree of the node, while the entry (u,v) and (v,u) will be set to -1. (The set will not allow an edge to be inserted twice and moreover the comparison logic in the explore method will not allow both the edge (u,v) and (v,u) to be inserted). All the other entries will contain a value of 0, which complies with the definition of the Laplacian matrix. After the Laplacian matrix is constructed its last row and last column is removed. This is achieved in algorithm 4. This algorithm copies each entry of the Laplacian matrix up to but not including the last index of each row and the last index of each column.

Afterwards the determinant of the matrix that results from the removal of the last row and last column is calculated in the determinant method (Algorithm 5). This method calculates the determinant of the matrix based on the Gaussian elimination method, which adds multiples of one row to another until all entries below the main diagonal are 0 and the determinant then is the product of the entries in the main diagonal (MIT OpenCourseWare).

The determinant is initialized to 1 and each pivot column is processed. If all the elements of the column are 0 it means that the matrix is singular[1], and the determinant is 0. This is handled properly in the code. If the pivot row is not the current row the rows are swapped, and the determinant is negated (Rule of linear algebra: when the rows of a matrix are swapped the determinant changes sign). Row reduction is performed to zero out entries below the pivot in the current column using row operations, maintaining the determinant's value up to a multiplicative constant. By multiplying all the diagonal elements after reduction, the method computes the determinant of the original matrix. The correctness follows from the fact that Gaussian elimination transforms the matrix to an upper triangular form, whose determinant is the product of its diagonal entries, and all operations preserve the determinant.

[1] a square matrix whose determinant is 0. When a row or column's elements in a matrix are all zeros, then the matrix is singular, as its determinant is zero.

3.Visualising one cheapest solution

An array of int values called prev is used to keep track of a path. Basically, the index of the array represents each node and for each node is recorded the node immediately before it on the path. This information is being updated in the explore method, each time a new node is discovered. The graph in this problem might consist of several connected components, for the starting point of each connected component the corresponding element in the prev array is 0 (which does not map to a particular node, since the representing values of our nodes start from 1). The previous node of a particular node is marked only when the node is first visited, therefore, we are sure that the edge is a tree edge.

As was proved above, when the cost of a bond is less than the cost of a bucket, the cheapest cost is achieved when a tree structure is obtained in each connected component. Therefore, by maintaining the path in this way, we are sure that we are storing a path that corresponds to the cheapest solution.

When the cost of a bond is greater than the cost of a bucket only disconnected nodes are displayed since the cheapest solution is the one in which all the dynos are disconnected and host themselves the bucket.

When the cost of a bond is equal to the cost of a bucket all the configurations that do not contain cycles correspond to the cheapest solution. The solution in which all the dynos are disconnected is displayed.

Time Complexity

The computeStatistics method is used to display a summary of the results: total cost of the cheapest solution, the number of cheapest solutions and a visualization of the cheapest solution.

In the best-case scenario (when the bucket cost is less than the bond cost) the if statement block will be executed (Algorithm 7, lines 2-4) and lines 11-13, each of which has a O(1) cost: performing a comparison, a multiplication and displaying these results.

In the worst-case scenario this method will call the DFS method. The overall time complexity of the DFS method without taking into consideration the call to the countSpanningTress method is O(n+ e) (S. Dasgupta, 2006). In the best case scenario, countSpanningTress methods will not be called: the block in lines 14-15 will be executed which has O(1) time complexity. Therefore, in the best case the time complexity of DFS is O(n+e), but in the worst case the countSpanningTrees method will be called for each connected component. The number of connected components might be a number from 1 up to n. So, the countSpanningTrees method can be called at most n times.

In the countSpaningTrees method, lines 4-9 have time complexity O (1), accessing an element of an array has constant time complexity. The for loop runs e (number of edges times). On the other hand, it calls the determinant and removeRowCol method. The removeRowCol method has O (n^2) time complexity since the outer for loop runs n (number of nodes times) and the inner loop runs n (number of nodes times) the body of the for loop has O(1) time complexity (accessing an element of the two-dimensional array). Therefore, the overall time complexity of the removeRowCol method is O (n^2). The determinant method on the other hand has a O (n^3) time complexity. The outer for loop in line 4 runs n (node times), the loop in line 6 runs at most n times, the for loop in line 18 runs n times, the innermost for loop in line 20 runs at most n times. Therefore, the body of the outer for loop runs at most n + n^2 times. The overall time complexity of the method is $O(n^3)$. Therefore, the overall time complexity of the countSpanningTrees method is

$$O(e) + O(n^2) + O(n^3) = O(n^3)$$

(In the worst case $e = n^2$).

Therefore, the time complexity of the DFS algorithm is:

$$O(n + e) + cO(n^3)$$

Where c is the number of connected components, but c can be as large as n. Therefore,

$$O(n + e) + cO(n^3) = O(n + e) + nO(n^3) = O(n^4)$$

Therefore, in the worst-case scenario computeStatistics method will have time complexity of $O(n^4)$.

If we take into consideration the time complexity of the method that reads the file, we know that it will run as many times as lines will process. The document has k + 1 lines, where k is the number of possible bonds. In the worst case $k = n^2$. Therefore, it has $O(n^2)$.

To conclude, the overall time complexity of the program is $O(n^4)$.

References

Cvetkovic, D., Rowlinson, P., & Simic, S. (2010). *An Introduction to The Theory of Graph Spectra*. Cambridge University Press. pp 184-193

MIT OpenCourseWare. (n.d.). *Evaluating the Determinant by Gaussian Elimination and by Row or Column Expansion*. Retrieved from https://ocw.mit.edu/ans7870/18/18.013a/textbook/HTML/chapter04/section03.html

S.Dasgupta,C.H.Papadimitriou,and U.V. Vazirani. (2006). *Algorithms.* pp. 87–94.