Solving the raspberry pi supercluster problem

Dibran Dokter s1047390 & Marnix Lukasse s1047400

This is our report for the first algorithms and data structures practical assignment. We decided to create the algorithm for the first problem, the Raspberry Pi Supercluster. We wrote our program in C++ since that is the language we are most experienced with.

In our paper we first go into the algorithm and its correctness. This part is broken up into the main parts of the algorithm. Namely the finding of the clusters, the finding of the longest path within each cluster and giving the answer to the question: how long will the longest path be when we connect those clusters in an optimal way? In each of these parts we will talk about how the algorithm works and prove its correctness. After this this part we go into the complexity of our algorithm.

Algorithm and Correctness

First we will go into the explanation of our algorithm. We begin by reading in the input, we first read the number of nodes and make that many nodes in a vector. After this we couple the nodes with their neighbours using a list of pointers. So node A gets a pointer to node B and the other way around. After we have read the input and created our node list we start working towards the solution.

Finding the clusters

Our first step is to put the nodes into clusters if they are connected. We do this by creating a list of booleans to check whether a node has already been added to a cluster or not. Then we find all the neighbours for every node and add them to the cluster and flag them as seen, we do this recursively for all the neighbours of the node. This comes down to using breadth-first search to discover all the nodes and putting them into a cluster. We do this for every node while checking if the node has not yet been added to a cluster. This results in a list of all the clusters.

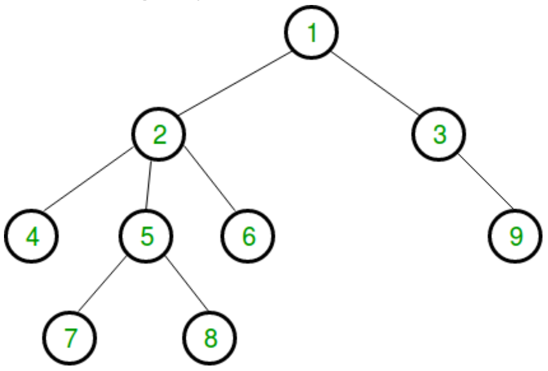
Finding the cluster with the longest path

After we have found all the clusters, we look for the longest cluster among them. To do this we find the length of the longest path in every cluster and return the cluster with the longest path.

For finding the longest path we take a random node from the cluster and calculate the longest path from there using breadth-first search. Then we run breadth-first search again from the node that was at the end of the longest path. The longest path we get from this second search is the longest path in the cluster.

This approach works since we can look at the cluster of nodes as if it is a n-ary tree. In this case we need to find the diameter of the tree to find the longest path. When we run breadth-first search from a random node in the tree we find the path to the leaf node of the deepest branch of the tree except for when the chosen node is in the longest branch, then we find the leaf node of the longest branch in the tree except the branch which the node is in. When we run breadth-first search from the leaf of the longest branch that we found, we find the longest path. This is because the longest path goes from the leaf in the deepest branch to the leaf in the deepest branch except for one through the root of the tree. Using this approach we always find this path.

As in the graph below we can see that if we choose any vertex in the left branch we will get the path to node 9 through the root when we perform breadth-first search. If we choose a node in the right branch we will always get path to node 7 or 8 through the root. When we run breadth-first search again from this node we will find the longest path.



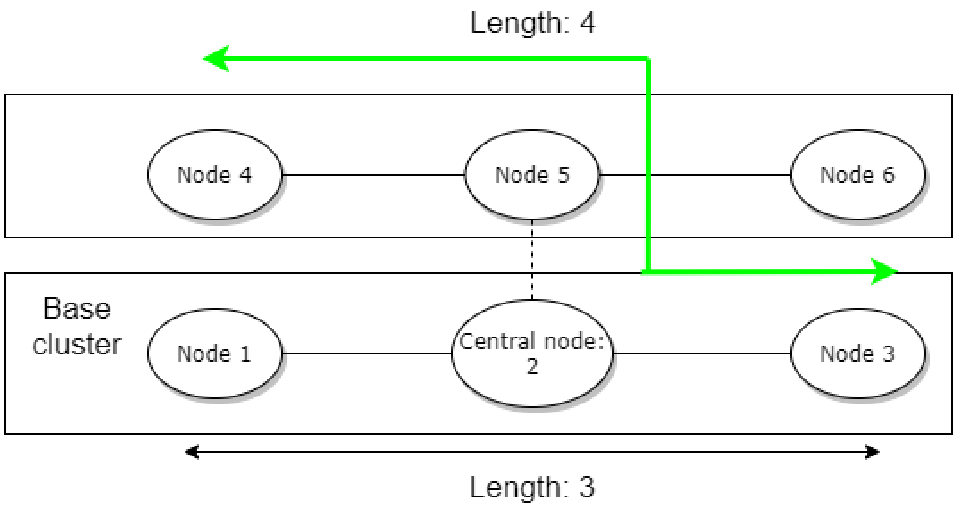
Finding the solution

Once the length of all the clusters is known, we actually have enough information to find the solution to the problem. We need a procedure to check what the effect on the longest path will be when we combine multiple clusters. As a starting point for this, the maximal length is the length of the cluster with the longest path. We made distinct cases for when this length is even or uneven, as the behaviour caused by adding smaller sized clusters varies between them. We describe these cases in detail below.

Note: in our report and code we mean the number of nodes when we speak about a longest path. So the path "1-2-3" would have a length of 3, the starting and end node are included. We realise the number of edges/hops is different. With the 'length' of a cluster we refer to the length of the longest path.

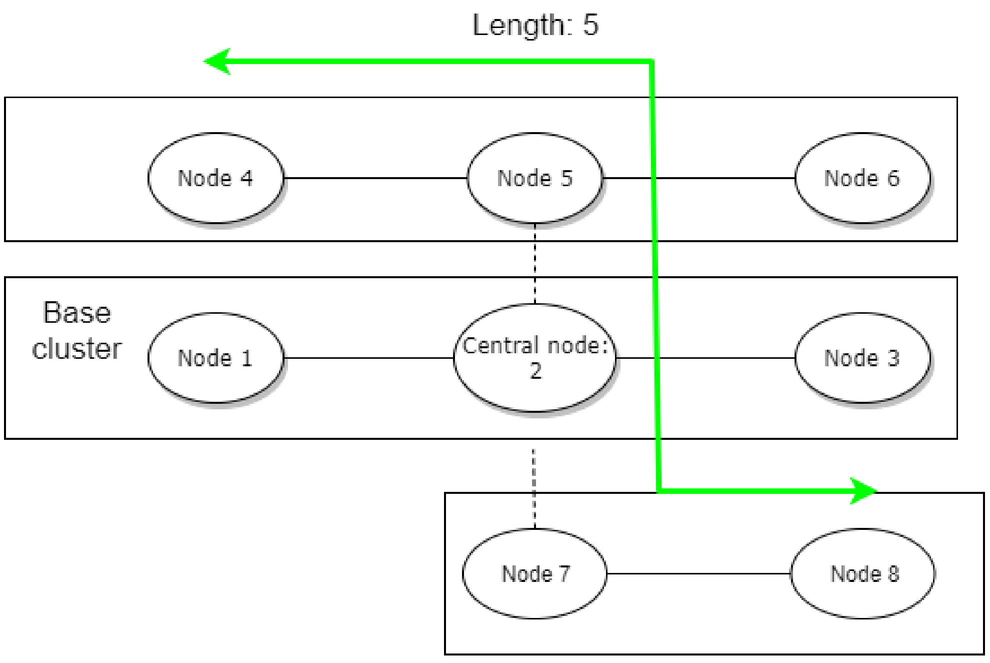
Uneven length

When the length of the longest path is uneven, there is always one middle node. This central (middle) node will be the point where the middle nodes of other clusters attach to during the merging of all clusters. The central node will always stay a valid middle node of the longest path. If we add another cluster with the same length, the longest path increases by one. There would also be another 'middle node', as the longest path now is an even number. We can see this in the following example:



After merging the central nodes of these clusters, node 5 could be called a central node too, but node 2 remains a central node as well so there's no need in changing our central node. We see how the length increases by one if one equally sized cluster is added to our base cluster. Note how adding a cluster of size 2 to the base cluster would mean the same for the length, in the picture node 6 is not needed to find a longest path. In the case we would have added a cluster of size 1 to the base cluster, the length would not increase at all. There is a pattern here, which we can formalize with the following rule: Given base cluster A with an even length n, adding a cluster B with smaller or equal length m to it by creating an edge between the central nodes of A and B will increase the length by exactly one if m > n-2. In our example, n was 3 so only clusters where m > 1 would increase the length by one. The reason for this is that such a cluster will have at least one path from its central node to one of its edge nodes of a length equal to the longest path from the central node in the base cluster to one of its edges. Now the extra edge between the two central nodes, causes the length to go up by one.

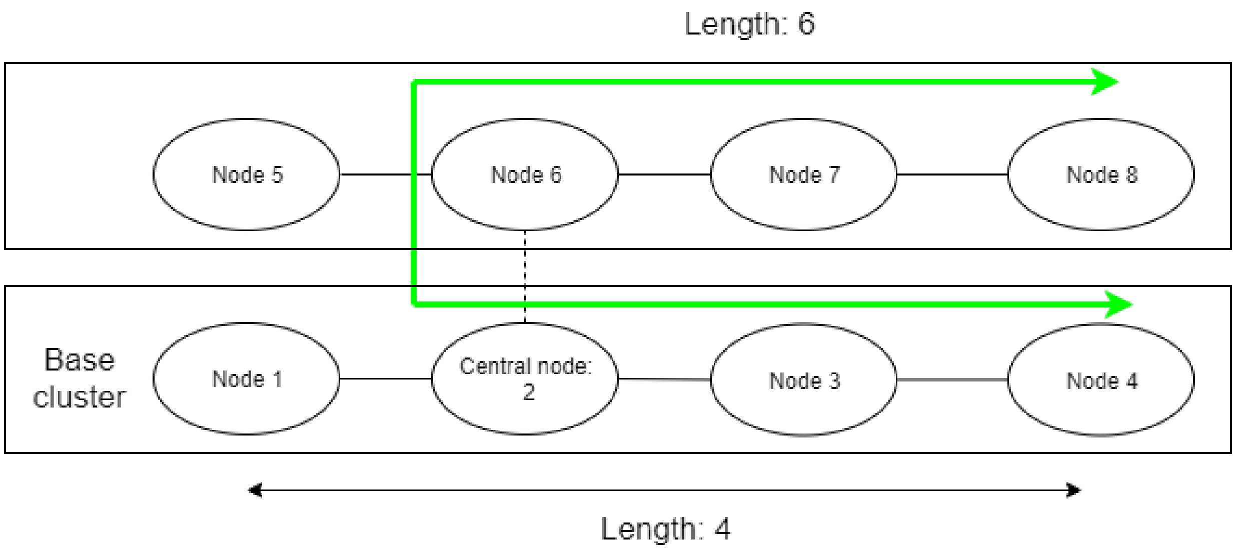
What happens if we don't add just one, but two or more clusters to the base cluster? In that case, the length could increase by a maximum of two, as we see in the example below.



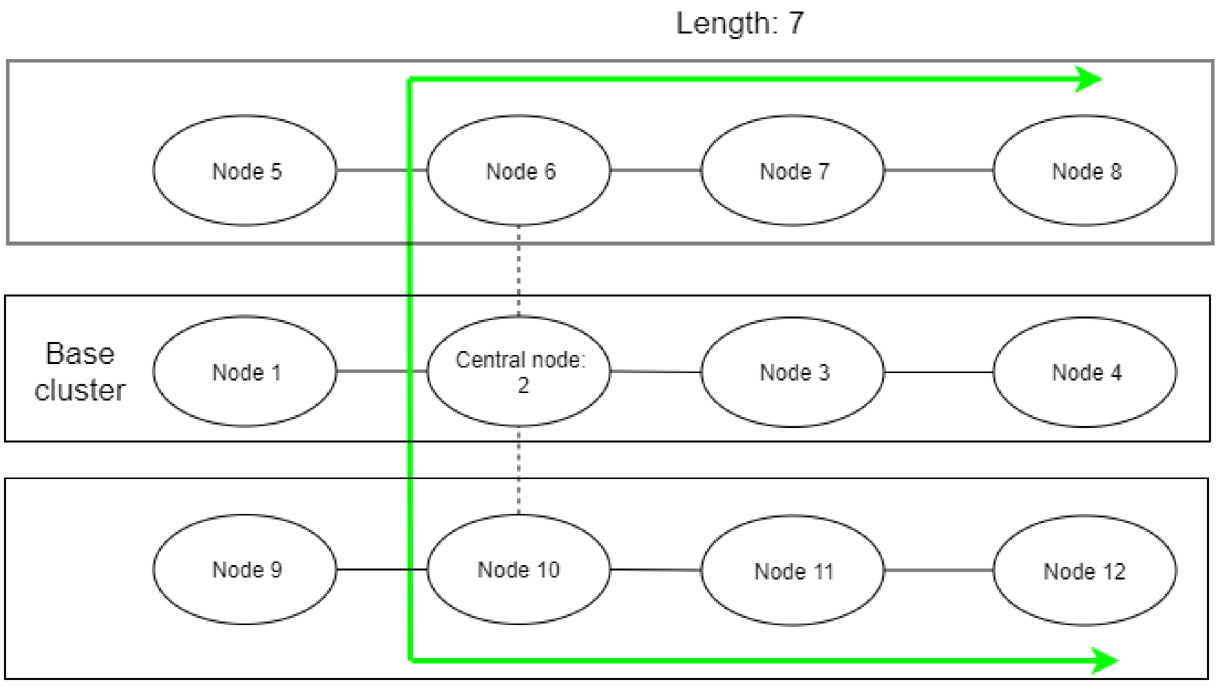
The same rule as described previously is used here to check whether the addition of this second cluster would increase the length of the total cluster. The extra hop from node 2 to node 7 increases the length by one again. Now any other smaller/equally sized clusters can't further increase the length. The reason for this is that we can have one transition from a cluster to the central node, and then one from the central node to another cluster, so at maximum 2 extra node visits by combining these clusters.

Even length

The case where the longest cluster is of even length is slightly more complicated. The reason for this is that there is some asymmetry. The central node has n nodes on one side, and n-1 nodes on the other side. So one of these paths from central node to one of its two edges is the longest. If we add another equally sized cluster to this base cluster, the length will increase by 2. The reason for this is the described asymmetry, the path follows both ‘longest’ sub-paths of each cluster. We can see the original length as (n-1 + 1 + n) = (2n), where 1 stands for the central node, n for the number of nodes on the larger side and n-1 for the other shorter side. After combining two clusters though, we now have (n + 1 + 1 + n) = (2n + 2) where both the 1’s represent the central nodes of each cluster, and n the longest sub-path from these clusters starting at the central node. The described situation is depicted below, n is 2 in this case. We see how the length increases by 2.

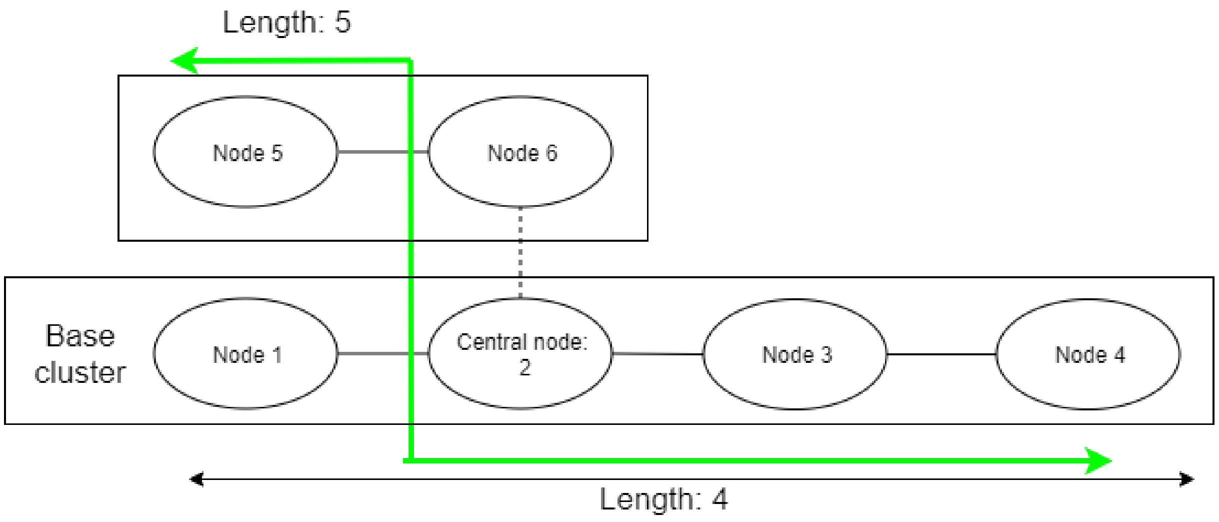


Adding any clusters now of smaller length (e.g. 3) will not change the length of the longest path. This is because the longest sub-path from its central node to an edge will always be smaller than the longest sub-path of the base cluster. Now what happens if we add another cluster of equal length? In that case the length increases by only one. The reason why it wouldn’t increase by 2 again is that the described asymmetry (taking longest sub-path twice) is already in place, so we aren’t able to gain length anymore by combining longest sub-paths. There is however still an extra transition from the central node to the newly added cluster, hence the increase by one.

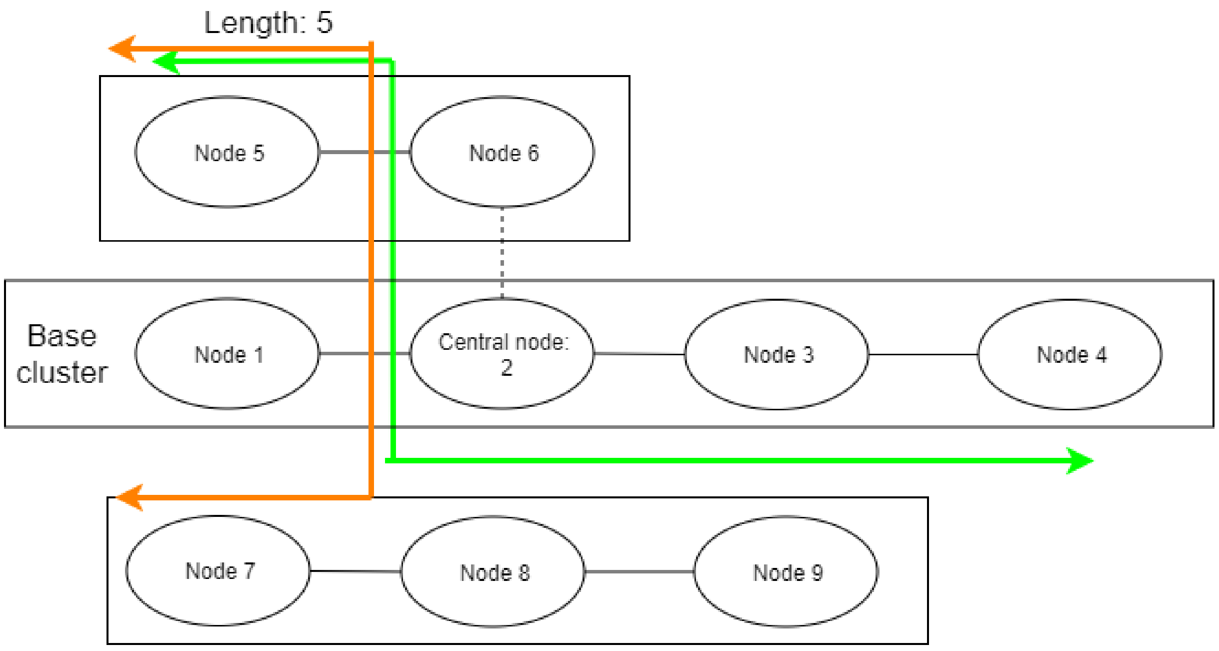


Any other clusters would not have an impact anymore, there is no way for them to increase this longest path.

What happens if we start with a base cluster of even length and we only add smaller-sized clusters? This case looks quite similar to the case with a base cluster of uneven length. Given base cluster A with length n and another cluster B with smaller length m, if m is greater or equal to (n-2) the size will increase by one. The reason is that cluster B will in that case still have a shortest sub-path (path from central node to one of the two leaves) equal to the shortest sub-path of cluster A. The extra connection between the central nodes, causes the length to increase by one.



Contrary to the uneven case, adding more cluster of smaller size will not have an effect on the longest path. The reason behind this is that the longest sub-paths of these clusters are always shorter than the longest sub-path of the base cluster. The extra connection between the central node and the 3rd cluster increases the length by one, but then having a sub-path which has at maximum still a sub-path that is at least one smaller than the sub-path of the base cluster decreases it by one again. These effects cancel each other out, as can be seen below.



Both the orange and green path have length 5, nothing changes by adding the cluster with nodes 7, 8 and 9.

Summarizing the even case: Given longest cluster A with length n, if there exists exactly one other cluster of equal length, total length increases by two. If there exist two or more clusters of equal length, the total length increases by three. If no clusters of equal length exist, if there exists any cluster with length greater or equal to n-2 the length increases by one.

Using these rules and behaviours that were described, we are able to quite quickly determine what the total length would be if we combine multiple clusters. These rules are very useful, as we don't have to actually combine the clusters into one which could be quite costly time-wise.

Complexity

For this algorithm we have found the complexity by looking at every part of the algorithm and determining the time complexity for that part. For this algorithm we first have to read the input, and then we find the clusters and determine the longest path. After this we find the solution using the algorithm described above.

Reading the input takes O(|E|) time since we need to connect all the nodes and thus need to do something for every edge in the input. We then have to create a number of vertices, that procedure is O(|V|). Finding the clusters takes O(|V|+|E|) since we run breadth-first search, which has a time complexity of O(|V| + |E|) for every cluster. We mark every node that we have added to a cluster as seen and ignore it if we see it again. Because of this we never handle a node twice. This gives us a time complexity for O(|V|+|E|). Finding the longest path takes O(|V|+|E|) since we run breadth-first search again only for two nodes in the cluster.

When we have this info we only need to find the solution as described under the heading "finding the solution". This part of the algorithm shouldn't take long, the complexity depends on the number of clusters. In worst case scenario where almost all clusters only contain a single node, the complexity approaches O(|V|).

In general this gives us the complexity of O(|V| + |E|) for the entire algorithm, which is linear time.

This complexity is assuming that we can access values in constant time. The access time depends on the data structure that is used for running breadth-first search. At first we used a map and a set, as those are ordered structures they gave us the access performance of O(log n). We noticed that this was a performance bottleneck, so we later substituted the use of those structures with an array. This gives accessing indices a time complexity of O(1).