“ Hub Location Network Optimization “

IMT Atlantique - Spring 2022

Miguel Ángel Guzmán Sánchez, Kadriye Nur Bakirci, Zohreh Kohandani
Shikhar Saini, Mahamat Nour Ali Mai

Table of Contents
In this report, you can see:

I. Presentation of our case study

In our logistic group we worked around the Hub Location Problem, which is one of the most discussed topics in location problems. The goal was minimizing the number of transportation links between origin and destination nodes by using hubs.

Since the costs of a hub network depend on its structure, by using fewer resources, demand pairs can be served more efficiently with a hub network than with a fully connected structure (full-mesh).

In this case we are provided with the following information: ˆ

- The location of the nodes is fixed ˆ
- Each spoke should be allocated to only one hub (single allocation) ˆ
- All the spokes should be allocated to a hub ˆ
- There is a fixed cost for locating a hub ˆ
- There is a variable cost, per the quantity flows, transferred through the network ˆ
- Transferring flow between hubs takes advantage of a discount factor (transferring the flow in big volumes and lower costs) ˆ
- Each hub has finite capacity.

II. Problem's modeling in the form of a MILP

The mixed-integer linear program (MILP) solver implements an LP-based branch-and-bound algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems.

In order to solve this problem we need to consider a few points as follow:

Sets and parameters

The sets and parameters of the problem are presented in the below table:

Sets:	
N	Set of vertices
$i, j \in N$	Indices of nodes ($i, j \in \{1, 2, \cdots, \|N\|\}$)
$k, l \in N$	Indices of hubs ($k, l \in \{1, 2, \cdots, \|N\|\}$)
Parameters:	
w_{ij}	Amount of flow from node i to node j
f_k	Fixed cost of locating a hub at node k
c_{ij}	Variable transfer cost of flow through the link from node i to node j
α	Hub-to-hub discount factor ($0 < \alpha < 1$)
C_k^{max}	Capacity of hub k
$O_i = \sum_{j=1}^{N} w_{ij}$	Total flow originating from node i
$D_i = \sum_{j=1}^{N} w_{ji}$	Total flow destination in node i

Decision variables

The main decisions in this problem are to first locate a number of hubs among the N possible number of nodes and then to allocate the remaining nodes to the located hubs. Therefore, the decision variables are defined as follows.

Ykl = 1 if arc (k; l) links two hubs; 0 otherwise.
The decision variable Ykl helps to design a tree hub-to-hub network.
Zik = 1 if the spoke i is allocated to the hub k; 0 otherwise.
When i = k, the variable Zik represents whether or not a hub is located at node k. The Z variables will be referred to as location/allocation variables.
Xikl = the amount of ow with origin in i 2 N traversing arc (k; l).
The X variables are independent low variables and their value is calculated based on the structural Y and Z variables.

Problem formulation

Objective function development, as it was mentioned earlier, is minimizing the cost by locating the hubs.

$$\min \sum_{k \in N} f_k Z_{kk} + \sum_{i \in N} \sum_{k \in N} (c_{ik} O_i + c_{ki} D_i) Z_{ik} + \sum_{i \in N} \sum_{k \in N} \sum_{l \in N, l \neq k} \alpha c_{kl} X_{ikl}$$

Constraints

a. Single Allocation

$$\sum_{k \in N} Z_{ik} = 1 \qquad \forall i \in N$$

b. Hub logical structure

$$Z_{kl} + Y_{kl} \leq Z_{ll} \qquad \forall k, l \in N, l > k$$
$$Z_{lk} + Y_{kl} \leq Z_{kk} \qquad \forall k, l \in N, l > k$$

c. Transit of flow through the tree

$$X_{ikl} + X_{ilk} \leq O_i Y_{kl} \qquad \forall i, k, l \in N, l > k$$

d. Flow conservation at each hub

$$O_i Z_{ik} + \sum_{l \in N, l \neq k} X_{ilk} = \sum_{l \in N, l \neq k} X_{ikl} + \sum_{l \in N} w_{il} Z_{lk} \qquad \forall i, k \in N, i \neq k$$

e. Capacity constraint

$$\sum_{i \in N} \sum_{l \in N} X_{ilk} + \sum_{i \in N} O_i Z_{ik} \leq C_k^{max} \qquad \forall k \in N$$

f. Tree structure

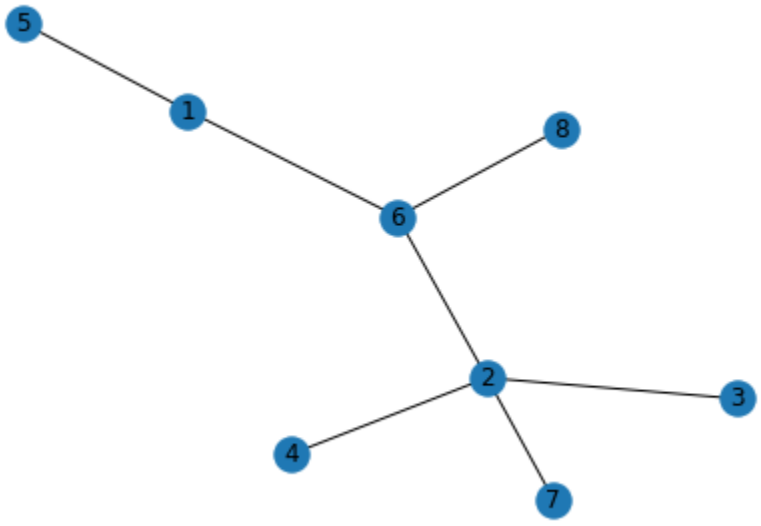
$$\sum_{k \in N} \sum_{l \in l} Y_{kl} = \sum_{k \in l} Z_{kk} - 1$$

g. Domain of variables

$$Z_{ik}, Y_{kl} \in \{0, 1\} \quad X_{ikl} \geq 0 \qquad \forall i, j, k \in N$$

III. The resolution of the “small” problem with MILP

In the small project for the value of MILP we obtained an objective function value of: 6794788.65.



Spanning tree solution representation for small dataset with MILP

IV. Choices for ILS meta-heuristics

Genotypic Representation (Prüfer Codes)

The most important element in the design of a decoder-based evolutionary algorithm is its genotypic representation. An evolutionary algorithm maintains a population of data structures that represent candidate solutions to a problem. In a decoder-based EA, each data structure can be thought of as providing instructions to a decoder that builds the solution the structure represents. The data structure is the solution's genotype; the decoded solution is its phenotype.

The decoder can consider problem-specific information such as constraints. Precisely, the most important reason why we chose the prüfer code as the genotypic representation of our solution is the fact that its decoding always guarantees the construction of a spanning tree, whereas other genotypic representations such as the list of edges do not always ensure a spanning tree and the performance of local search an perturbation operations have to be done with extreme care.

Local Search Operator

The local search operator we utilized for our ILS implementation was the one proposed by the skeleton implementation provided by the professor. This operation consists in applying reversion moves on the prüfer code elements.

A reversion move is done reversing the order of elements between positions *p1* and *p2* of the prüfer code digits. In our implementation we chose to evaluate all the possible *p1* and *p2* pairs, also called 'neighbors', for each local search iteration instead of doing a limited number of reversion moves since the computation time was fast enough to allow this exploration without delaying the perturbation operations very much.

The total number of neighbors for a m-lenght prüfer code (m is 'n' number of nodes -2) is defined by:

$\frac{m!}{(n-k)!k!}$, $where\ m\ is\ the\ number\ of\ pr\ddot{u}fer\ code\ digits\ and\ k\ the\ number\ of\ positions\ to\ choose\ (2\ -\ tuple)$

So for the small dataset we evaluate the following number of neighbors per local search iteration:

$$\frac{m!}{(n-k)!k!} = \frac{6!}{(6-2)!2!} = 15$$

Whereas for the small dataset we evaluate the following number of neighbors per local search iteration:

$$\frac{m!}{(n-k)!k!} = \frac{28!}{(28-2)!2!} = 378$$

This operator proved to be very efficient in finding local minima.

Perturbation Operator

Originally, we tried the perturbation operator proposed by the skeleton implementation provided by the professor. It consisted of a swapping perturbation that performed a single swap move on the solution (changing the elements at positions p1 and p2).

This was operator did not work as a good perturbator since it constrained the set of possibly explored solutions to a set of maximum $(n-2)!$ solutions (case where all the prüfer digits are different in the initial random solution). Considering that the solution space is made up of n^{n-2} solutions, the ratio of maximum number of solutions that can be explored against the solution space ($\frac{(n-2)!}{n^{n-2}}$) is extremely low. Moreover, the nodes designed as hubs would always be the same without any change given the prüfer decoding algorithm.

To give a practical example, we could get an initial random prüfer Code for the small dataset such as the following:

[6, 3, 2, 1, 4, 0]

This is the case where all the random prüfer digits produce no repeated values stated above.

So given that set of numbers the maximum possible number of prüfer Code permutations which the maximum possible number of spanning trees that can be evaluated would be:

$$(n-2)! = (8-2)! = 6! = 720$$

And for the small dataset the solution space is made up of the following number of spanning trees:

$$n^{n-2} = 8^6 = 262,144$$

So, if we compute the ratio of the maximum number of solutions that can be explored against the solution space for this particular example of the small dataset we get:

$$\left(\frac{(n-2)!}{n^{n-2}}\right) = \left(\frac{(8-2)!}{8^{8-2}}\right) = 0.0027 = 0.027\%$$

Our perturbation proposal had to be able to exit the aforementioned constraint of possible solutions to explore, which always kept the same prüfer digits for all perturbated solutions. The proposed solution consisted then of randomly replacing one of the prüfer digits with a different randomly generated prüfer digit. In this way we could add, delete or reshape the connections to the hubs with a very simple perturbation that could computationally be done/undone very easily. In this way we made sure to explore a completely different set of solutions with the local search after each perturbation.

**The perturbation algorithm will be explained with more detail in the pseudocode description section.*

V. Description of the implementation

Use of ‘Memory’ concept for ILS

The perturbations may depend on any of the previous s∗. In this case one has a walk in S^* with memory. In practice, much of the potential complexity of ILS is hidden in the history dependence. If there happens to be no such dependence, the walk has no memory: the perturbation and acceptance criterion do not depend on any of the solutions visited previously during the walk, and one accepts or not $s^{*'}$ with a fixed rule. This leads to random walk dynamics on S^* that are “Markovian”, the probability of making a particular step from s_1^* to s_2^* depending only on s_1^* to s_2^* . Most of the work using ILS has been of this type, though recent studies show unambiguously that incorporating memory enhances performance.

Incorporating memory proved to be a critical step for maximizing the performance of our ILS algorithm.

Perturbation with Memory Implementation Pseudocode

1. Select a random value from 0 to n-3 index to change and store it in a variable.
2. Select a random value from 0 to n-1 to insert on the random index.
3. Check that the random number is different from the number on the random index. If it is the same, repeat the second step. Otherwise, continue.
4. Save the random number in a variable.
5. Create a tuple containing: (random index, random number).
6. Check if the tuple already exists in a global hashing structure (set). If so, repeat from the first step.
7. Store the tuple in a global set.
8. Perform the perturbation operation: replace the random index’s value with the value of the random number.

Functions for the perturbation of solutions

In order to be able to have memory for all the performed perturbations over a single current best solution, first we need to abstract a perturbation operation as a data structure that can ‘encapsulate’ the description of the operation and be saved inside a ‘storing’ data structure.

A perturbation operation is composed of:

1) an index of a digit to be replaced in the solution
2) a digit to replace the given element of the solution

The chosen data structure to encapsulate the perturbation elements was a simple 2-tuple.

These two elements were saved inside the 2-tuple as first-index element and second-index element respectively.

As for the ‘storing’ data structure the Python in-built ‘set’ was used since it is implemented as a hashing structure that guarantees an 0(1) element-lookup time, so we could check in constant time if a given perturbation has already been performed or not.

VI. The resolution of the “small” and the “big” problem with ILS

1. The small problem

As it was mentioned in part III Objective value HUB Network Optimization Problem, as it was expected, turned out to be 6794788.65.
The global minimum of the ILS is 6482028.75.
The prüfer code representation of the solution for the small dataset is:

[1, 5, 0, 5, 1, 5]

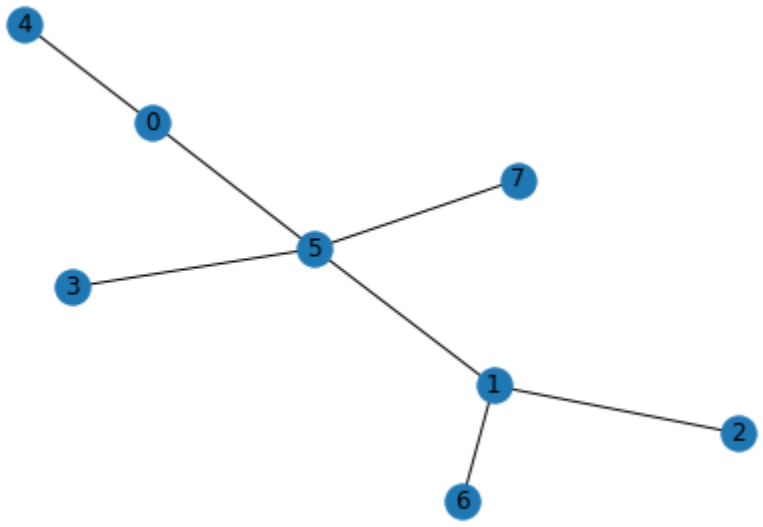


Figure 5.1 Spanning tree solution representation for small dataset with ILS

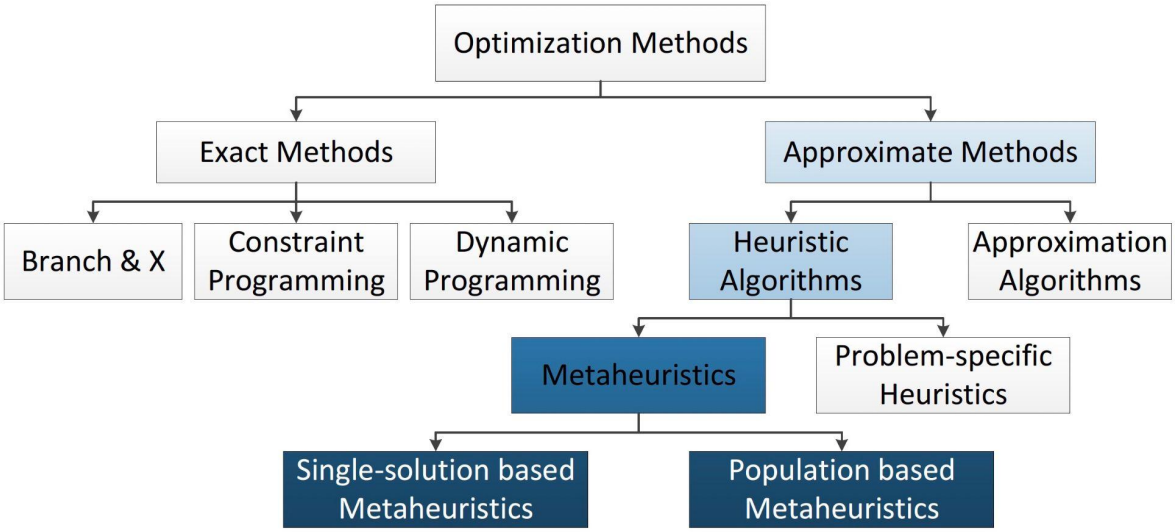
Therefore we can calculate the precision of the metaheuristic solution against the deterministic solution objective function value:

6482028.75 x 1.048 = 6482028.75
In other words our precision factor is 104.8%

This is very interesting because it means that the MILP deterministic solution apparently does not find the most optimal solution for the small dataset even though it is really close.

2. The big problem

Since we were dealing with a large data set, using an Exact method was practically impossible. Therefore, we needed to use one of the approximate methods to generate high quality solutions in a reasonable time for empirical purposes. However, there is no guarantee for finding a global optimal solution. Thus, for the big data set the global minimum is unknown.



In this study, we took the Single Solution Based Metaheuristic approach.

The ILS input parameters for this case study

In this section we present the tuning values that maximized the performance of our algorithm for both datasets.

For the small dataset

- **ALPHA = 0.05** # defines the ratio in which a worse explored perturbated solution can be set as a next step
- **MAX_ITER = 400** # maximum number of iterations
- **MAX_ITER_NI = 200** # number of iterations without improvement of the objective function

- **MAX_ITER_LS = 5** # maximum number of iterations of the local search operator (Outer loop)

For the large dataset

- **ALPHA = 0.025**
- **MAX_ITER = 1000**
- **MAX_ITER_NI = 300**
- **MAX_ITER_LS = 2**

In this study we took Iterated local search (ILS), because in this metaheuristic method one iteratively builds a sequence of solutions generated by an embedded heuristic, leading to far better solutions than if one were to use repeated random trials of that heuristic. Although normally the better the local search, the better the corresponding ILS, but if we assume that the total computation time is fixed, it might be better to apply a faster but less effective local search algorithm than a slower and more powerful one. [1]

The stopping criterion of ILS

1. The maximum number of iterations is achieved which is equivalent to MAX_ITER
2. The maximum number of non-improvements is reached, which is MAX_ITER_NI
3. The maximum number of perturbations that can be done over a single best solution

The first 2 stopping criteria are very common amongst most ILS metaheuristics, while the third one is particularly dependent on our implementation of the *memory* concept.

The maximum number of perturbations that can be done over a single best solution is defined by the following formula:

$$(n - 1) * (n - 2)$$

$$Where\ (n - 1)\ defines\ the\ number\ of\ digits\ that\ can\ be\ chosen\ to\ be\ a\ pr\ddot{u}fer\ digit,$$
$$and\ (n - 2)\ defines\ the\ length\ of\ a\ pr\ddot{u}fer\ code\ both\ for\ n\ nodes.$$

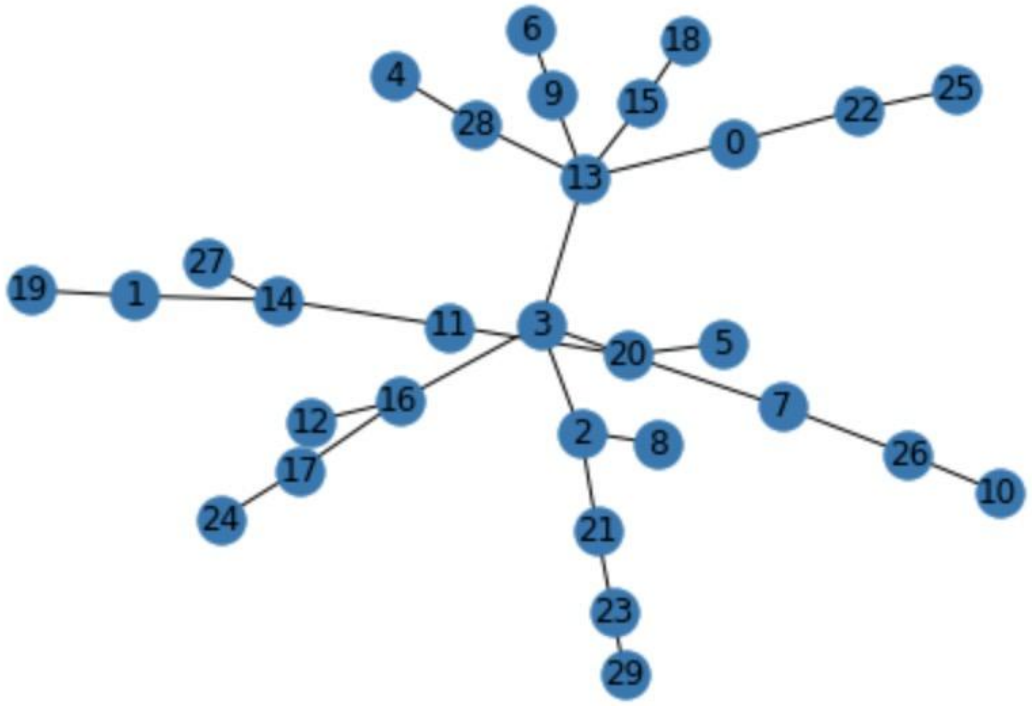
This stopping criterion is crucially important because it prevents the algorithm from entering an infinite loop in the case where all the possible perturbations for a single solution have already been explored so the algorithm knows that the storing set is already full and does not keep generating additional perturbations until infinity.

VII. A small study on different ILS input parameters

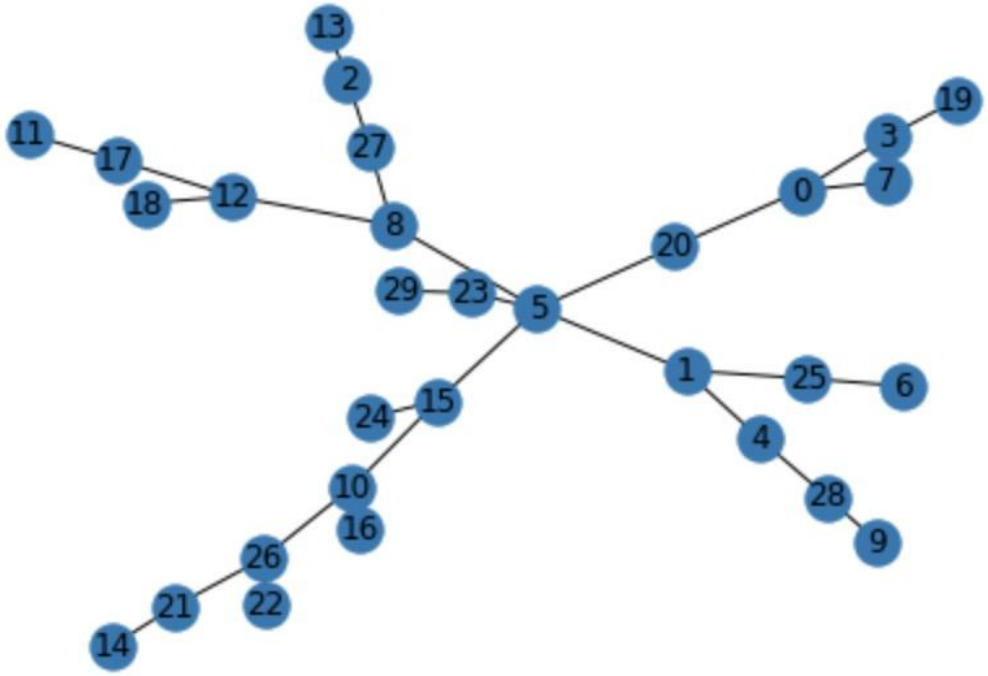
In our study we ran the code with several ILS input parameters. The table below shows the results for the large 30-node dataset.

No	Number of Iterations	Cost	ALPHA	MAX_ITER	MAX_ITER_NI	MAX_ITER_LS
1	103	2,05E+08	0.05	1000	200	3
2	72	2,47E+08	0.05	1000	200	3
3	90	2,08E+08	0.05	1000	200	3
4	174	2,11E+08	0.05	1000	200	3
5	143	2.40E+08	0.05	1000	260	3
6	150	2,26E+08	0.05	1000	260	3
7	134	2,32E+08	0.05	1000	260	3
8	201	2,02E+08	0.05	1000	260	3
9	98	2,11E+08	0.05	1000	260	3
10	160	2,21E+08	0.05	1000	260	3
11	819	2,05E+08	0.025	1000	300	2
12	**470**	**1,74E+08**	**0.025**	**1000**	**300**	**2**
13	89	2,17E+08	0.05	1000	260	3
14	610	2,04E+08	0.025	1000	300	2
15	**924**	**1,80E+08**	**0.025**	**1000**	**300**	**2**

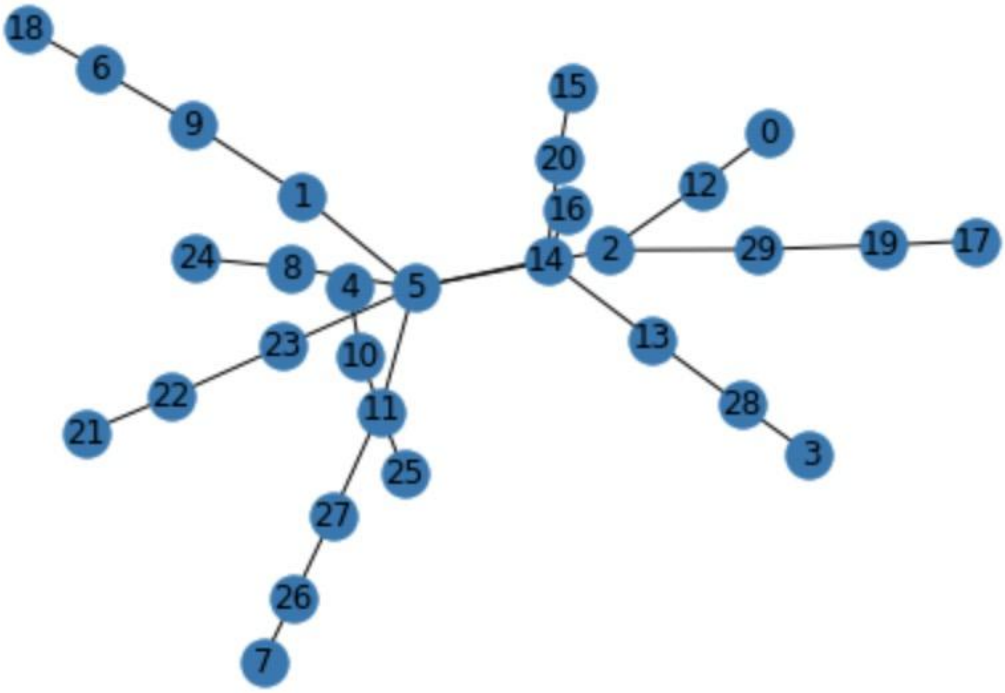
1. The image of spanning tree for the first set of parameters.



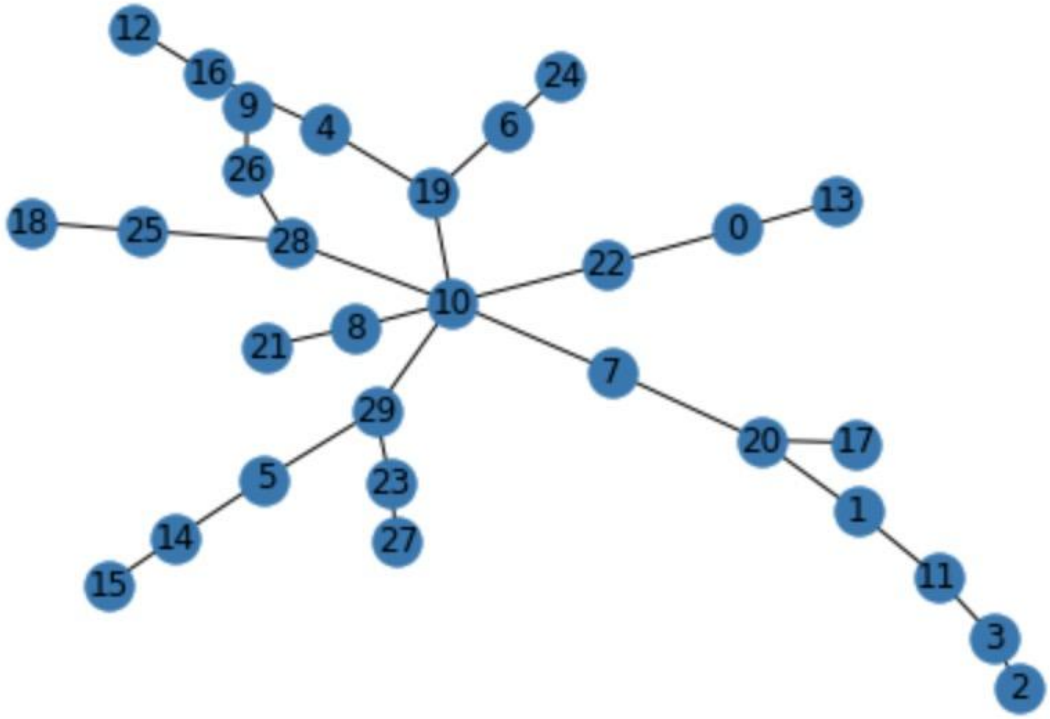
2. The image of spanning tree for the second set of parameters.



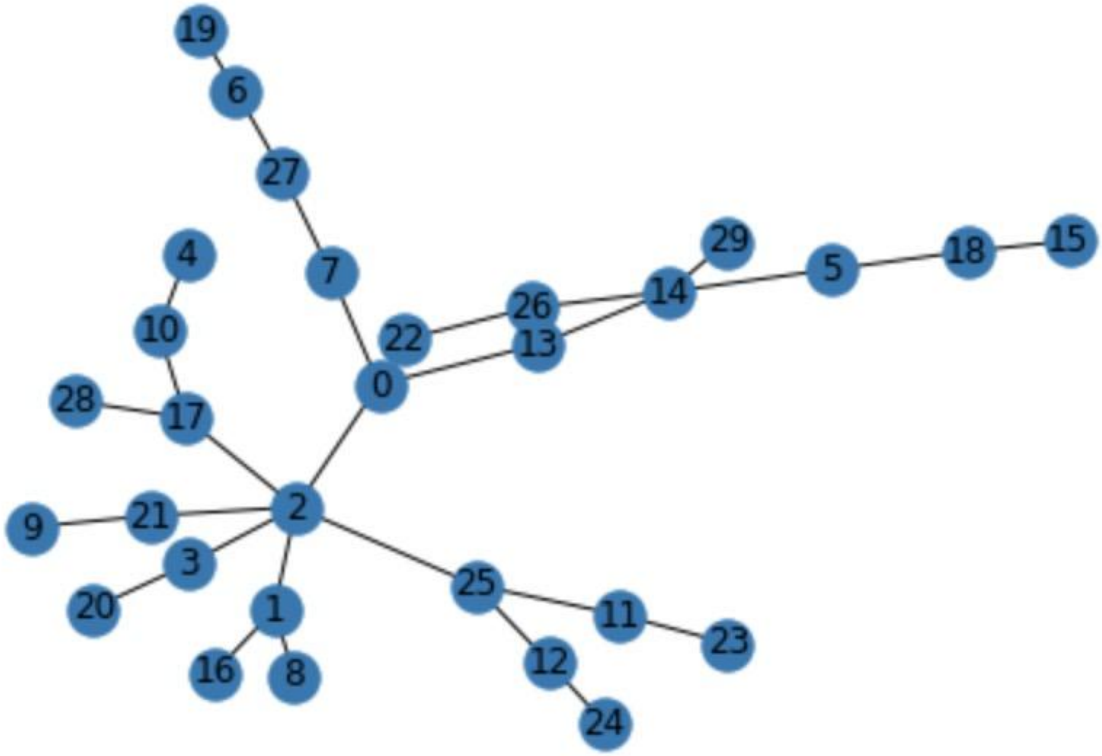
3. The image of spanning tree for the third set of parameters.



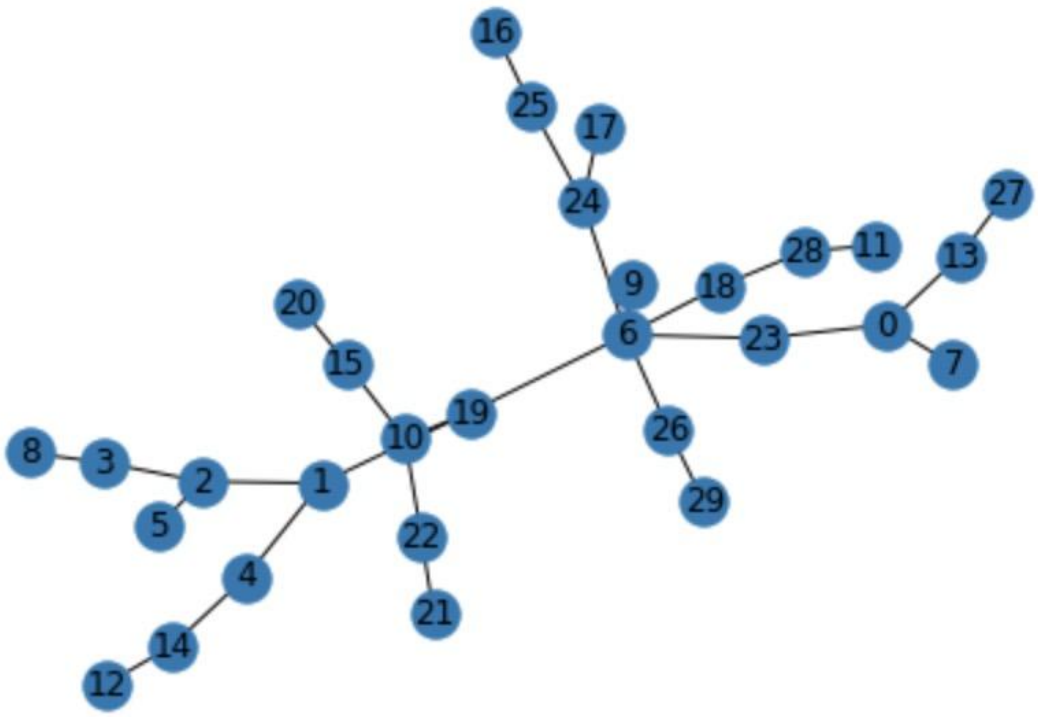
4. The image of spanning tree for the 4th set of parameters.



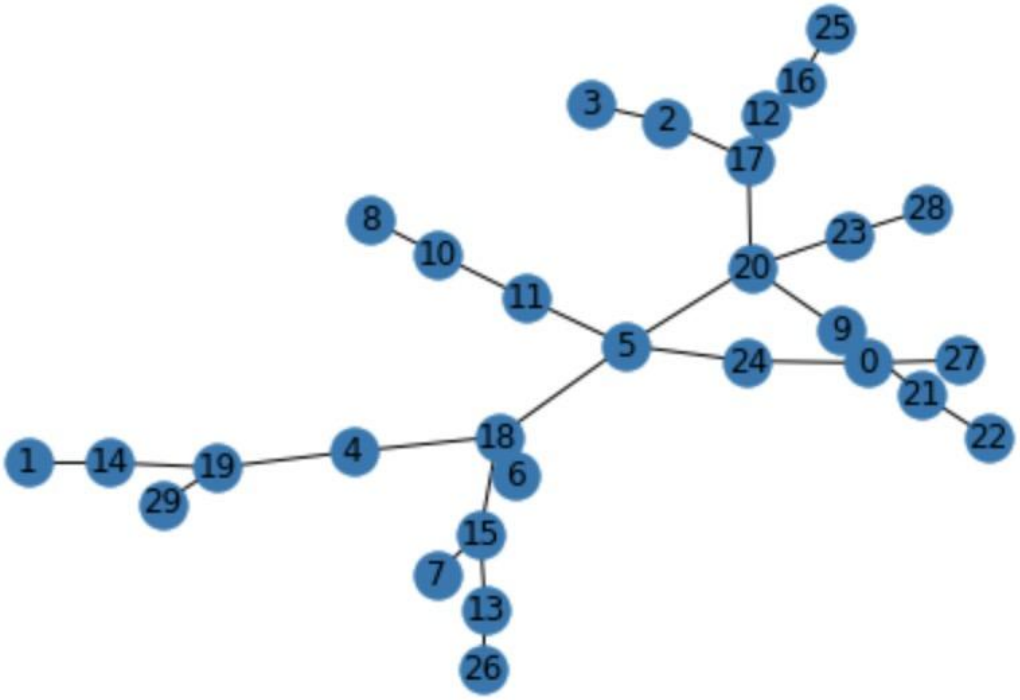
5. The image of spanning tree for the 5th set of parameters.



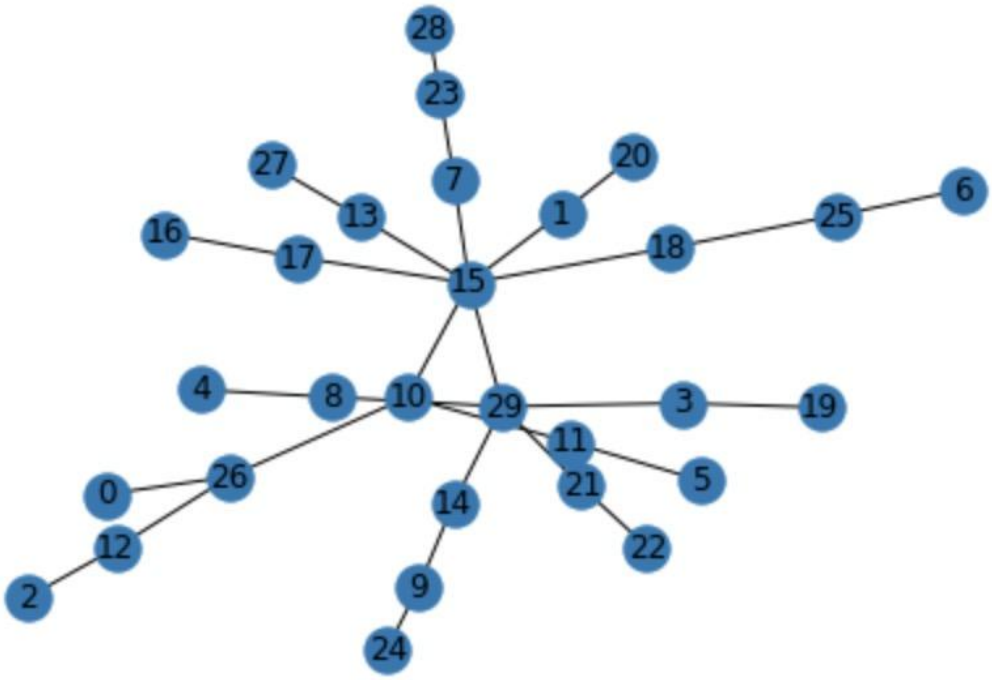
6. The image of spanning tree for the 6th set of parameters.



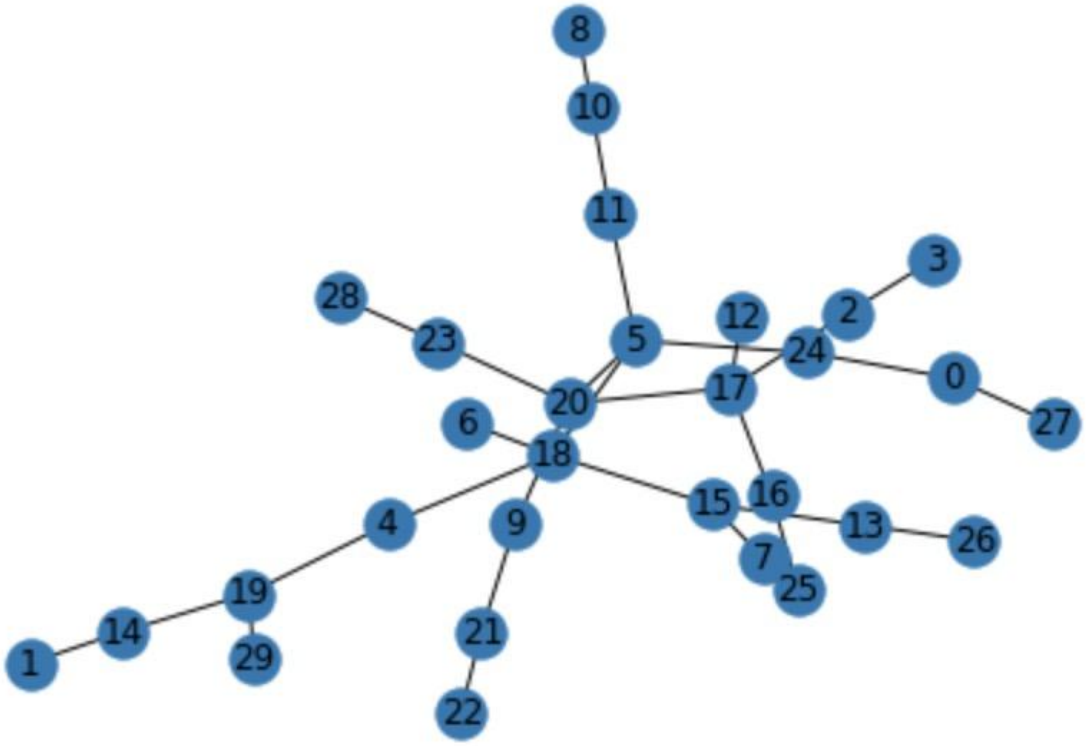
7. The image of spanning tree for the 7th set of parameters.



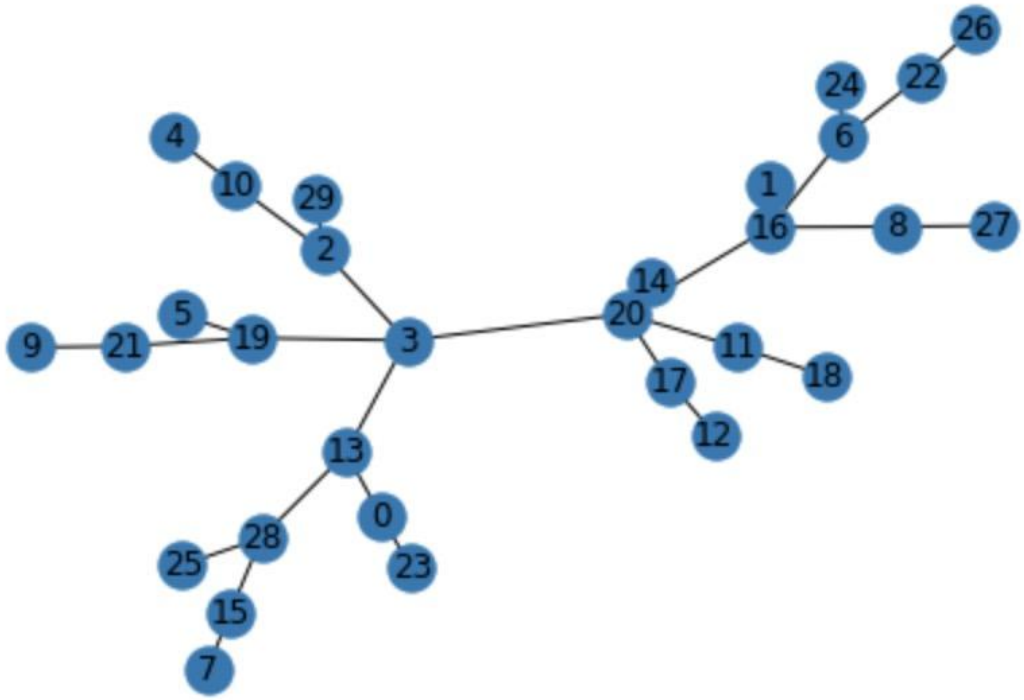
8. The image of spanning tree for the 8th set of parameters.



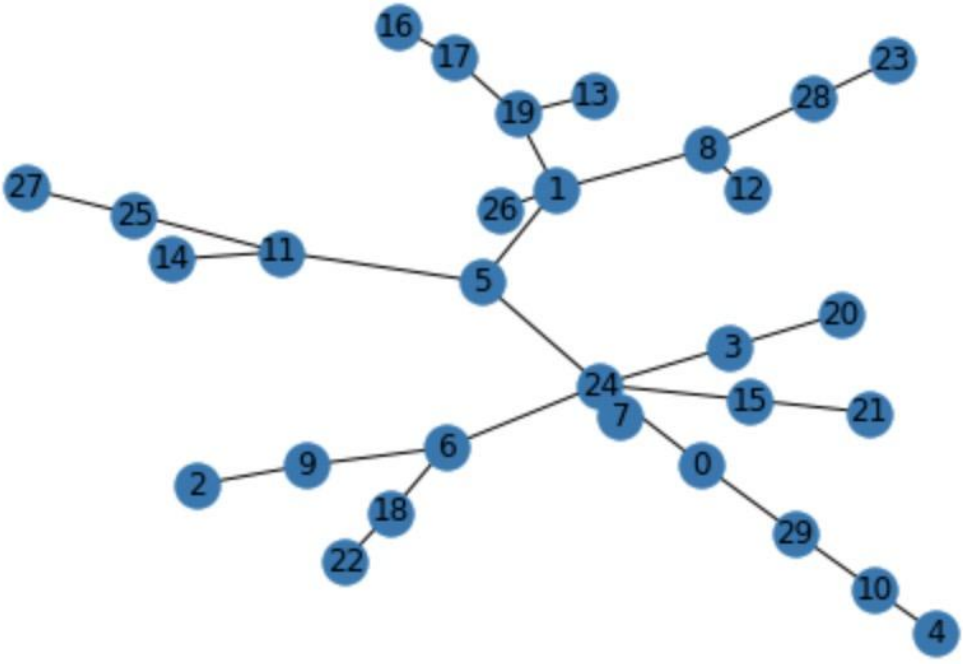
9. The image of spanning tree for the 9th set of parameters.



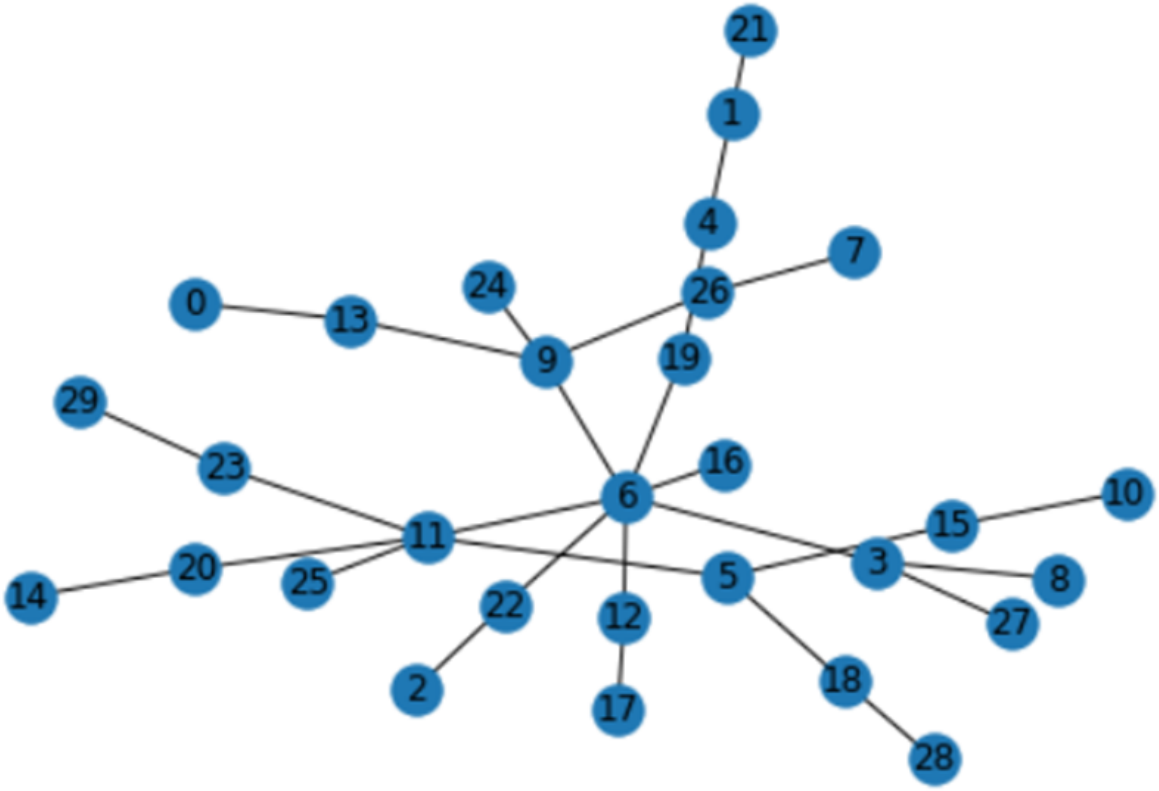
10. The image of spanning tree for the 10th set of parameters.



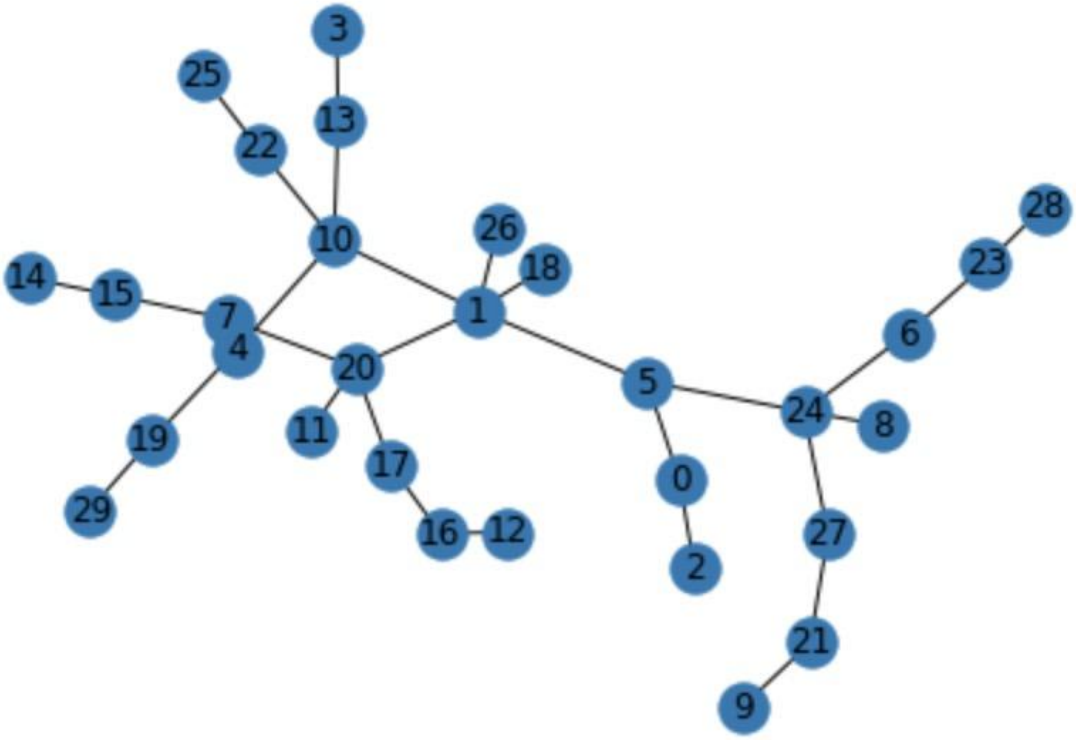
11. The image of spanning tree for the 11th set of parameters.



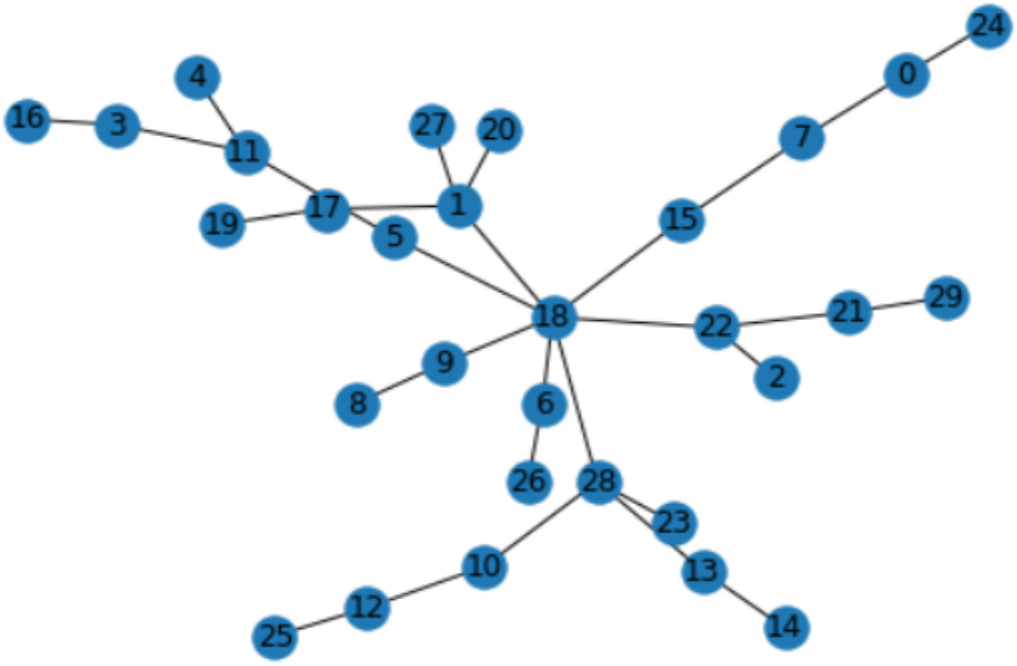
12. The image of spanning tree for the 12th set of parameters.



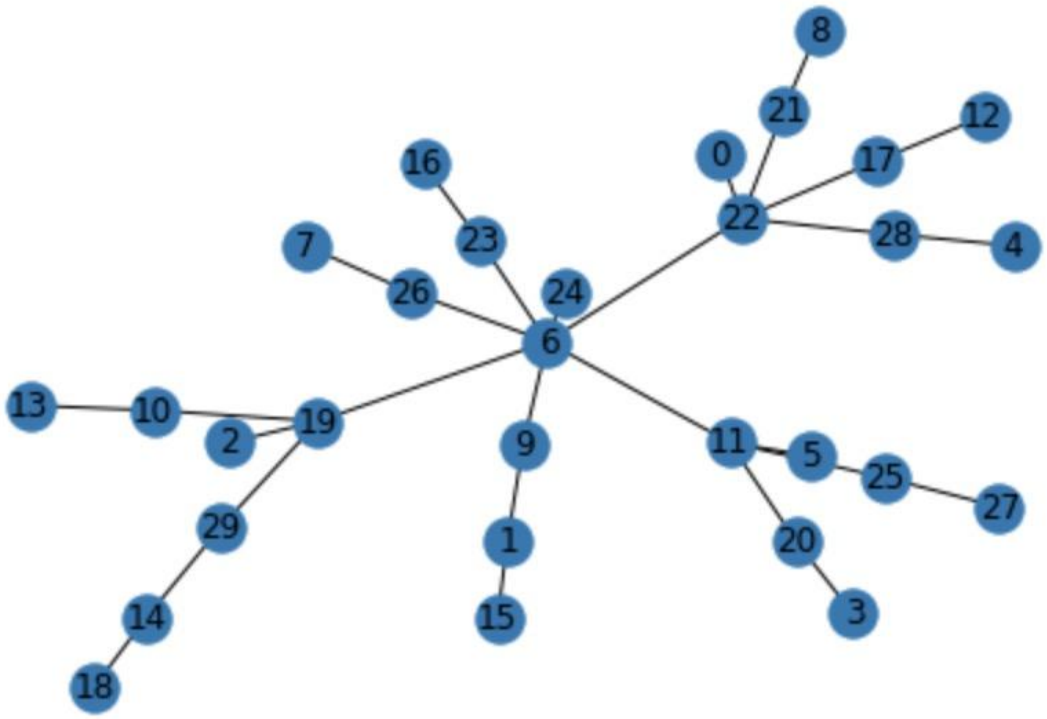
13. The image of spanning tree for the 13th set of parameters.



14. The image of spanning tree for the 14th set of parameters.



15. The image of spanning tree for the 12th set of parameters for the second time, to make sure the parameters are set accurately.



The final exact results obtained front the first try and the second try of 12th set of parameters are as follow:

The first try results:

```
After 470  ILS iterations, the best solution is:
[13, 22, 26, 3, 15, 9, 20, 5, 6, 12, 6, 11, 1, 4, 19, 6, 6, 9, 11, 9, 6, 3, 6, 11, 18, 5, 11, 23]
with total cost: [1.74008295e+08]
```

The second try results:

```
After 924  ILS iterations, the best solution is:
[22, 19, 20, 28, 11, 26, 21, 17, 10, 19, 1, 9, 6, 23, 22, 14, 29, 11, 22, 6, 6, 6, 25, 11, 6, 22, 6, 19]
with total cost: [1.79650135e+08]
```

Last but not least, in this study we used Google Colab and Jupyter notebook to run the codes.

References

- [1] A Beginner’s Introduction to Iterated Local Search - MIC’2001 - 4th Metaheuristics International Conference
- Prüfer Numbers: A Poor Representation of Spanning Trees for Evolutionary Search
- Hub Location Network Optimization - OR Nice Organizers - IMT Atlantique
- IMT atlantique - Operational research - slides from lecture 03