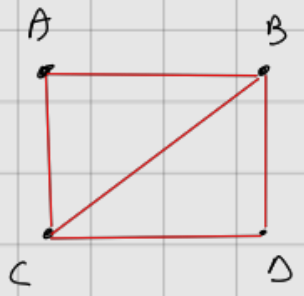
	x	y
A	707	47
B	925	317
C	796	770
D	32	823

	A	B	C	D
A	-	1	1	-
B	1	-	1	1
C	1	1	-	1
D	-	1	1	-



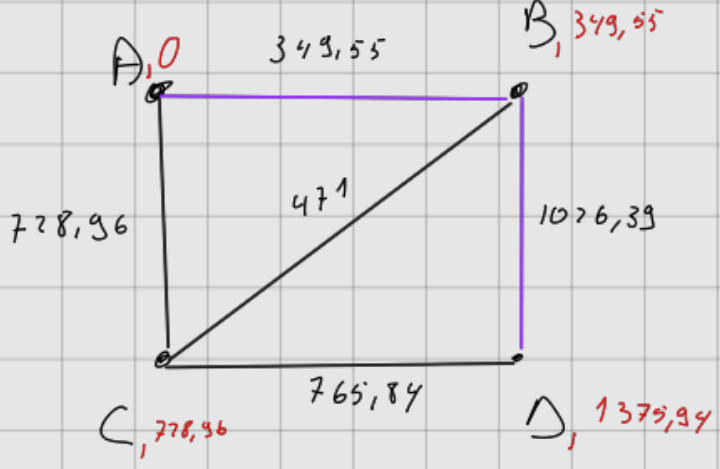
$$distance = ((x_2-x_1)^2 + (y_2-y_1)^2)^{0,5}$$

a)

	A	B	C	D
A	0	349,55	728,96	0
B	349,55	0	471	1026,39
C	728,96	471	0	765,84
D	0	1026,39	765,84	0

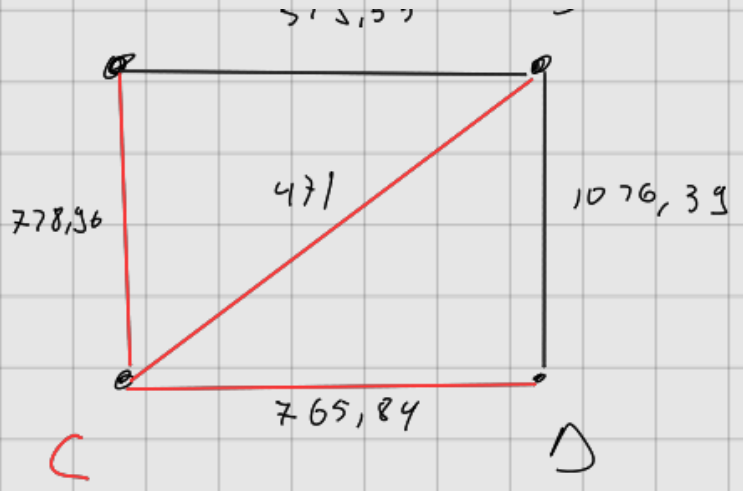
b) A - D

A = 0



Shortest path A -> D = 1375,94 (A -> B -> D)

c) A B

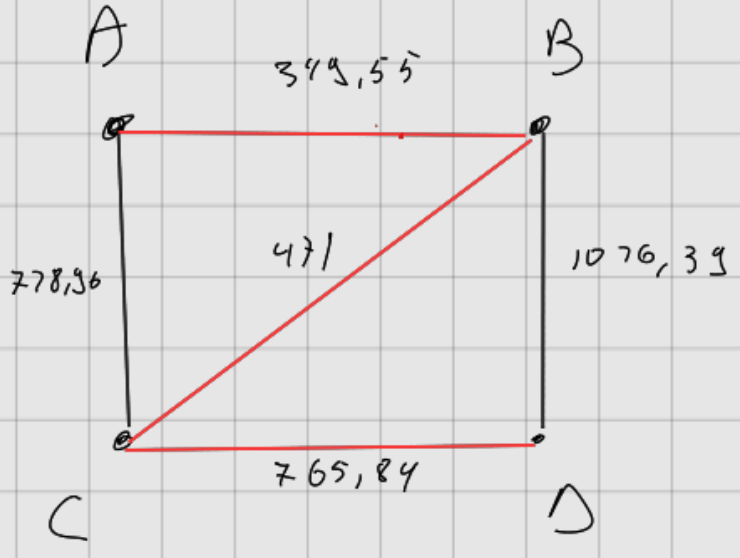


1) Random point C

2) Closest node: B (471)
3) Closest node: A (778,96)
4) Closest node: D (765,84)

Min spanning tree = 1965,8

d)



1) Smallest length = AB (349,55)

2) Add BC

3) Add CD

Min spanning tree = 1586, 39

```
import numpy as np
import csv


def fetchcsv(filename):
    # using the NumPy library to read a CSV file and convert it into a NumPy
array of integers.
    r = np.genfromtxt(filename, delimiter=",", dtype=int, names=True)

    #
    # This code is reading a CSV file and converting it into a numpy array of
strings.
    reader = csv.reader(open(filename, "r"), delimiter=",")
    x = list(reader)
    result = np.array(x).astype("str")

    Labels = []
    # Delete the first element of the array which is the labels
    for i in range(len(result[0])):
        Labels.append(result[0][i])
    del Labels[0]

    dictionary = {}

    keys = []
    values = []

    # retrieves the corresponding values from the Labels list and appends them
to the keys list.

    for i in range(len(r[0]) - 1):
        keys.append(Labels[i])

    temp_dic = {}

    sub_keys = []

    sub_values = []

    # Iterate over the rows of the matrix
    for j in range(len(r[0]) - 1):
        # Iterate over the rows of the matrix
        for i in range(len(r[0]) - 1):
            # Check if the value at position (j, i+1) is not equal to -1
            if r[j][i + 1] != -1:
                # Append the label associated with the column index to
sub_keys
                sub_keys.append(Labels[i])
                # Append the value at position (j, i+1) to sub_values
```

```
                sub_values.append(r[j][i + 1])
        # Create a dictionary from sub_keys and sub_values using zip() and
assign it to the temp_dic variable
        for key, value in zip(sub_keys, sub_values):
            temp_dic[key] = value
        # Append the temp_dic dictionary to the values list
        values.append(temp_dic)

        sub_values = []
        sub_keys = []
        temp_dic = {}

    # For each iteration, it assigns the value of the
    # `value` variable to the key of the `key` variable in the `dictionary`.
Finally, it returns the
    # `dictionary`.
    for key, value in zip(keys, values):
        dictionary[key] = value

    return dictionary


# inputs csv file and returns a dictionary
graph = fetchcsv("inputfile AS4.csv")
print(graph)


def dijkstra(graph, start, goal):
    shortest_distance = {}
    predecessor = {}
    unseenNodes = graph
    infinity = 9999999
    path = []

    # This is the first step in the Dijkstra's algorithm for finding the
shortest path in a graph.
    for node in unseenNodes:
        shortest_distance[node] = infinity
    shortest_distance[start] = 0
    # This code initializes a while loop that will continue until the
`unseenNodes` dictionary is empty.
    while unseenNodes:
        minNode = None
        # This code is finding the node with the shortest distance from the
start node among the nodes that
        # have not been visited yet. It iterates over all the nodes in the
`unseenNodes` dictionary and checks
        # if the `minNode` variable is `None`. If it is, it assigns the
current node to `minNode`. If it is
```

```
        # not `None`, it compares the `shortest_distance` of the current node
with the `shortest_distance` of
        # the `minNode`. If the `shortest_distance` of the current node is
less than the `shortest_distance`
        # of the `minNode`, it assigns the current node to `minNode`. At the
end of the loop, `minNode` will
        # contain the node with the shortest distance from the start node
among the nodes that have not been
        # visited yet.
        for node in unseenNodes:
            if minNode is None:
                minNode = node
            elif shortest_distance[node] < shortest_distance[minNode]:
                minNode = node
        # This code is updating the `shortest_distance` and `predecessor`
dictionaries for each neighboring
        # node of the current `minNode` in the Dijkstra's algorithm. It
iterates over the items in the
        # dictionary of the `minNode` node in the `graph` dictionary, where
each item represents a neighboring
        # node and its weight. For each neighboring node, it checks if the sum
of the weight of the edge
        # between the `minNode` and the neighboring node and the
`shortest_distance` of the `minNode` is less
        # than the current `shortest_distance` of the neighboring node. If it
is, it updates the
        # `shortest_distance` of the neighboring node to this sum and sets the
`predecessor` of the
        # neighboring node to the `minNode`. This process continues until all
nodes in the `graph` have been
        # visited and their `shortest_distance` and `predecessor` values have
been updated.
        for childNode, weight in graph[minNode].items():
            if weight + shortest_distance[minNode] <
shortest_distance[childNode]:
                shortest_distance[childNode] = weight +
shortest_distance[minNode]
                predecessor[childNode] = minNode
        # Removes the `minNode` from the `unseenNodes` dictionary. This
        # is because the `minNode` has been visited and its shortest distance
from the start node has
        # been calculated, so it is no longer "unseen".
        unseenNodes.pop(minNode)

    currentNode = goal
    # This code is tracing back the shortest path from the `goal` node to the
`start` node using the
```

```
    # `predecessor` dictionary that was created during the Dijkstra's
algorithm. It starts with the `goal`
    # node and iteratively inserts the current node at the beginning of the
`path` list and updates the
    # `currentNode` variable to its predecessor node until it reaches the
`start` node. If a node does not
    # have a predecessor in the `predecessor` dictionary, it means that there
is no path from the `start`
    # node to that node, so the code raises a `KeyError` and prints "Path not
reachable". Finally, the
    # `start` node is inserted at the beginning of the `path` list, and if the
`goal` node is reachable
    # from the `start` node, the code prints the shortest distance and the
path.
    while currentNode != start:
        try:
            path.insert(0, currentNode)
            currentNode = predecessor[currentNode]
        except KeyError:
            print("Path not reachable")
            break

    # This code is checking if the `goal` node is reachable from the `start`
node by checking if the
    # `shortest_distance` of the `goal` node is not equal to infinity. If the
`goal` node is reachable, it
    # inserts the `start` node at the beginning of the `path` list using the
`insert()` method with index
    # 0. Then, it prints the shortest distance from the `start` node to the
`goal` node and the path from
    # the `start` node to the `goal` node using the `print()` function.
    path.insert(0, start)
    if shortest_distance[goal] != infinity:
        print("Shortest distance is " + str(shortest_distance[goal]))
        print("And the path is " + str(path))


dijkstra(graph, "a", "b")
```