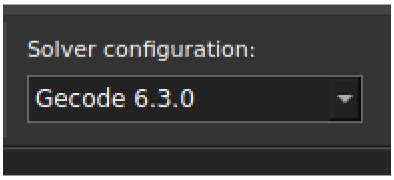
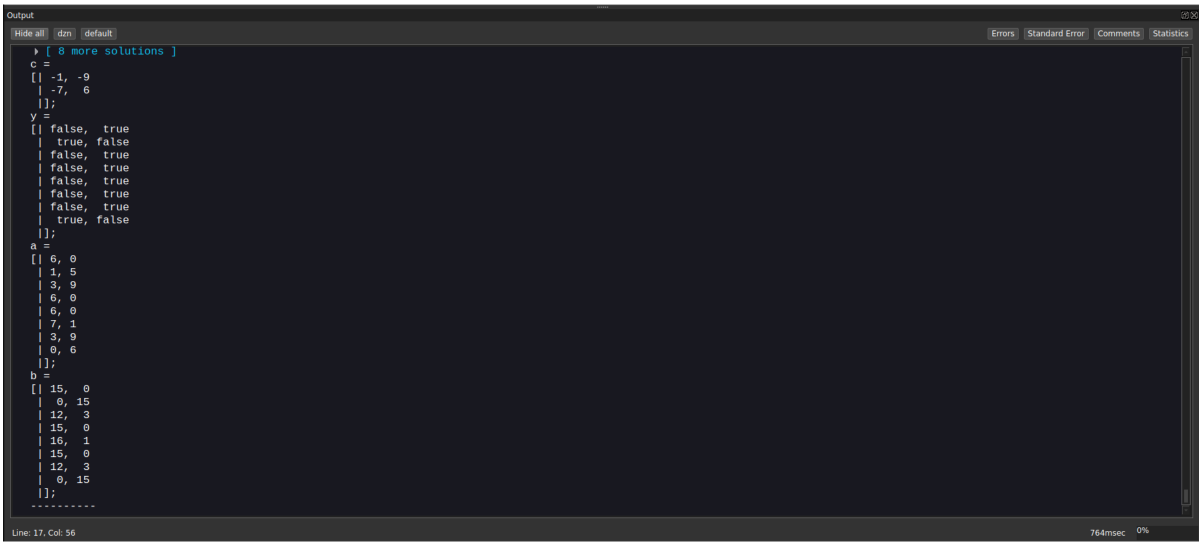
C : is representing the coordinates of the cluster centroids
Y: is showing use which node belonges to which cluster centroid
A and B : are for when we want the problem be linear and not to use abs

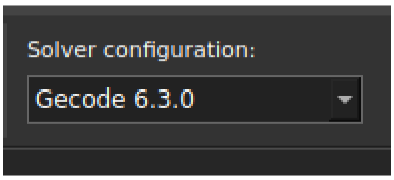
1) Minimizing the distance of each node to its cluster.
We must make sure that each cluster centroid has at least one member.
We must make sure that each cluster centroid has at most ‘n’ members(capacity).
All cluster centroids must be different.




```
int: k=2;
int: n=8;
array[1..k,1..2] of var -1000.0..1000.0 :c;
array[1..n,1..k] of var bool:y;

array[1..n,1..2] of var -1000.0..1000.0 :N =[|-7, 6,
                              |-2, -9,
                              |2, 3,
                              |-7, 6,
                              |-7, 7,
                              |-8, 6,   |2, 3,
                              |-1, -9,
                              |];

array[1..n,1..k] of  var -1000.0..1000.0: a;
array[1..n,1..k] of  var -1000.0..1000.0: b;


constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])>=0);
constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])<=n);

constraint forall(i in 1..n)(sum(j in 1..k)(y[i,j])=1);


constraint forall(i in 1..n)(forall(j in 1..k )(abs(N[i,1]-c[j,1])=a[i,j]));

constraint forall(i in 1..n)(forall(j in 1..k )(abs(N[i,2]-c[j,2])=b[i,j]));


solve minimize sum(i in 1..n)(sum(j in 1..k)((a[i,j]+b[i,j])*y[i,j]));
```



2)Maximizing the distance between clusters while minimizing the distance of each node to its cluster.



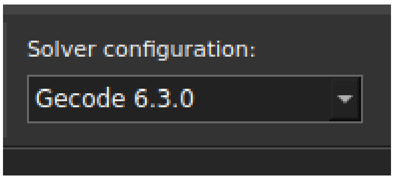
```
int: k=2;
int: n=3;
array[1..k,1..2] of var int :c;
array[1..n,1..k] of var 0..1:y;

array[1..n,1..2] of int :N =[|-7, -9,
                             |-6, -9,
                             |2, 3,

                             |];

constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])>=1);
constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])<=n);

constraint forall(i in 1..n)(sum(j in 1..k)(y[i,j])=1);
constraint forall(i in 1..n)(forall(j,p in 1..k where p!=j)((abs(N[i,1]-c[j,1])+abs(N[i,2]-c[j,2]))*y[i,j] <= (abs(N[i,1]-c[p,1])+abs(N[i,2]-c[p,2])) ));


constraint forall(i,ii,iii in 1..n where i!=ii /\ i!=iii)(forall(j in 1..k)((abs(N[i,1]-c[j,1])+abs(N[i,2]-c[j,2]))*y[i,j] <= max(abs(N[i,2]-N[iii,2])+abs(N[i,1]-N[iii,1]),abs(N[i,2]-N[ii,2])+abs(N[i,1]-N[ii,1]))));



solve maximize sum(j in 1..k)(sum(p in 1..k where p!=k)(abs(c[j,2]-c[p,2])+abs(c[j,1]-c[p,1])));
/*  output["result:",show(sum(j in 1..k)(sum(p in 1..k)(abs(c[j,2]-c[p,2])+abs(c[j,1]-c[p,1]))))];*/
```

Output

```
c =
[| 22,  3
 | -7, -9
 |];
y =
[| 0, 1
 | 0, 1
 | 1, 0
 |];
----------
==========
Finished in 1s 63msec.
```

Line: 10, Col: 32   1s 63msec   0%

In jupiter:



```
%%minizinc -m bind
%-m bind, mapea las variables de minizinc a variables de python (ejemplo, queens)

include "globals.mzn";
include "alldifferent.mzn";
int: k=2;
int: n=8;
array[1..k,1..2] of var int :c;
array[1..n,1..k] of var bool:y;

array[1..n,1..2] of int :N =[|-7, 6,
                            |-2, -9,
                            |2, 3,
                            |-7, 6,
                            |-7, 7,
                            |-8, 6,
                            |2, 3,
                            |-1, -9,
                            |];

array[1..n,1..k] of var int: a;
array[1..n,1..k] of var int: b;

constraint all_different(c);
constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])>=1);
constraint forall(j in 1..k)(sum(i in 1..n)(y[i,j])<=n);

constraint forall(i in 1..n)(sum(j in 1..k)(y[i,j])=1);


constraint forall(i in 1..n)(forall(j in 1..k )(abs(N[i,1]-c[j,1])=a[i,j]));

constraint forall(i in 1..n)(forall(j in 1..k )(abs(N[i,2]-c[j,2])=b[i,j]));


solve minimize sum(i in 1..n)(sum(j in 1..k)((a[i,j]+b[i,j])*y[i,j]));
```

```
c
```

[21] ✓ 0.0s

```
[[-2, -9], [-7, 6]]
```

```
y
```

[22] ✓ 0.0s

```
[[False, True],
 [True, False],
 [False, True],
 [True, False],
 [False, True],
 [False, True],
 [False, True],
 [True, False]]
```

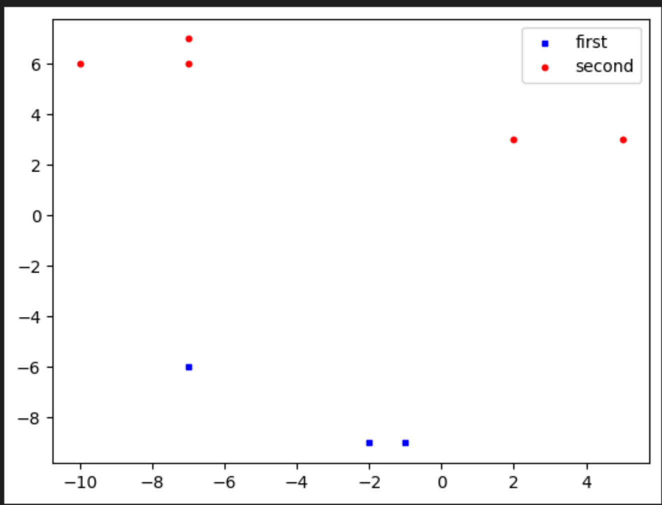
```
print(Y)
```

[44] ✓ 0.0s

```
{2: [-7, 5, -7, -10, 2], 1: [-2, -7, -1]}
{2: [6, 3, 7, 6, 3], 1: [-9, -6, -9]}
```

```
#plot
fig = plt.figure()
ax1 = fig.add_subplot(111)

ax1.scatter(X[1],Y[1], s=10, c='b', marker="s", label='first')
ax1.scatter(X[2],Y[2], s=10, c='r', marker="o", label='second')
plt.legend(loc='upper right')
plt.show()
```

[43] ✓ 0.3s



Phase 2:

First we choose the first k elements of the N list and then we consider them be in different clusters so it is better to choose them randomly which in phase 3 we did. Since N elements are random there is no need. Then they get there demands from source ‘s’. Then we have n nodes to got from k elements with capacity 1. The costs are the distance between n elements to k clusters.

```

N=[[6.846075241173382, -0.7061717170771983],
 [-5.883076343963735, 6.276749557836347],
 [-6.10955557850388, 5.107993849741389],
 [8.104350235417453, -0.3171652778124078],

]
k=2
n=len(N)
```

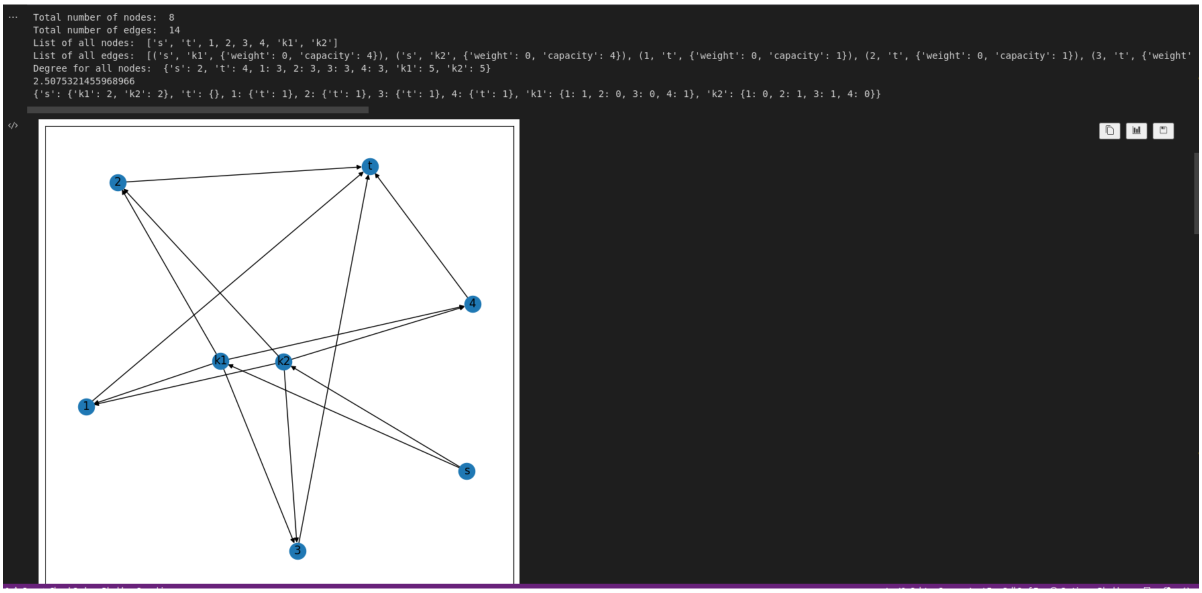
[5] ✓ 0.0s

Minimum cost flow problem with source ‘s’ and sink ‘t’:

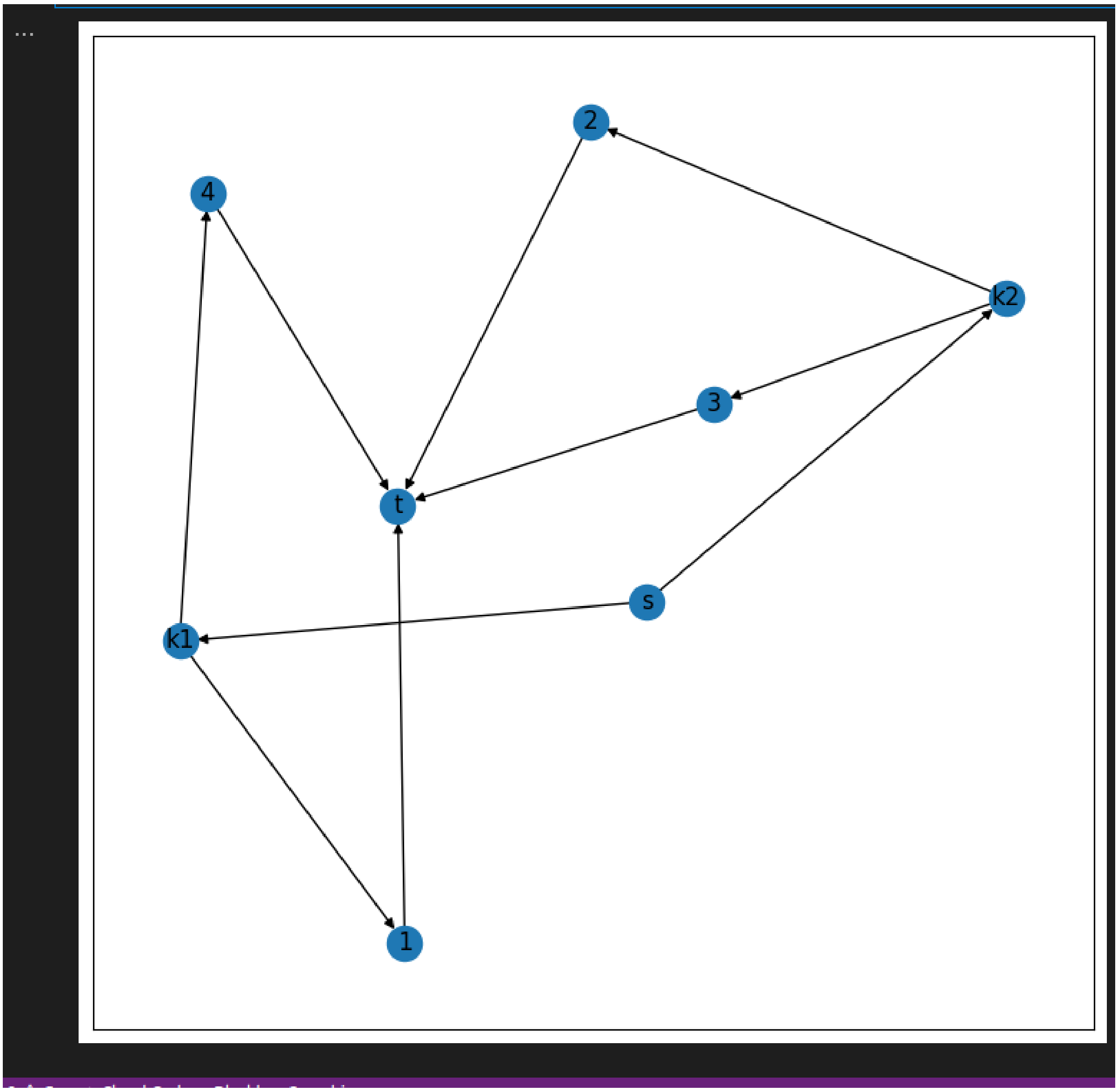
```
Total number of nodes: 8

Total number of edges: 14
```

List of all nodes: ['s', 't', 1, 2, 3, 4, 'k1', 'k2'] List of all edges: [('s', 'k1', {'weight': 0, 'capacity': 4}), ('s', 'k2', {'weight': 0, 'capacity': 4}), (1, 't', {'weight': 0, 'capacity': 1}), (2, 't', {'weight': 0, 'capacity': 1}), (3, 't', {'weight': 0, 'capacity': 1}), (4, 't', {'weight': 0, 'capacity': 1}), ('k1', 1, {'weight': 0.0, 'capacity': 1}), ('k1', 2, {'weight': 14.518694487075585, 'capacity': 1}), ('k1', 3, {'weight': 14.200453907328109, 'capacity': 1}), ('k1', 4, {'weight': 1.3170352960074336, 'capacity': 1}), ('k2', 1, {'weight': 14.518694487075585, 'capacity': 1}), ('k2', 2, {'weight': 0.0, 'capacity': 1}), ('k2', 3, {'weight': 1.1904968495894628, 'capacity': 1}), ('k2', 4, {'weight': 15.463758119337228, 'capacity': 1})]

Degree for all nodes: {'s': 2, 't': 4, 1: 3, 2: 3, 3: 3, 4: 3, 'k1': 5, 'k2': 5} 2.5075321455968966 {'s': {'k1': 2, 'k2': 2}, 't': {}, 1: {'t': 1}, 2: {'t': 1}, 3: {'t': 1}, 4: {'t': 1}, 'k1': {1: 1, 2: 0, 3: 0, 4: 1}, 'k2': {1: 0, 2: 1, 3: 1, 4: 0}}



And finally the optimal solution only:



Phase 3:
Running phase 2 code for each k ,x times(10 here) with random number of ks chosen from nodes in N.

```python
]
def phase2(N,k):
    G=nx.DiGraph()
    G.add_node("s", demand=-1*n)
    G.add_node("t", demand=n)

    G.add_nodes_from(range(1,n+1))

    G.add_nodes_from(['k'+str(i) for i in range(1,k+1)])

    for i in range(1,k+1):
        G.add_edge("s",'k'+str(i),weight = 0 ,capacity=n)

    for i in range(1,k+1):
        for j in range(1,n+1):
            G.add_edge('k'+str(i),j,weight = cost(N[i-1],N[j-1]) ,capacity=1)

    for j in range(1,n+1):
        G.add_edge(j,"t" ,weight = 0 ,capacity=1)

    flowCost,flowDict=nx.capacity_scaling(G)
    return flowCost

def newN(NN,k):
    nodes= copy.copy(NN)
    selectedNodes=set()
    while(len(selectedNodes)<k):
        selectedNodes.add(random.randint(0,n-1))
    newNodes=[]
    temp=[]
    for i in range(n):
        if i not in selectedNodes:
            newNodes.append(nodes[i])
        else:
            temp.append(nodes[i])
    return temp+newNodes

flowCostsForAllClusters=[0 for i in range(n)]
minFlowCosts=set()
for i in range(10):
    for k in range(2,n+1):
        flowCostsForAllClusters[k-2] = phase2(newN(N,k),k)
    minFlowCosts.add( flowCostsForAllClusters.index(min(flowCostsForAllClusters)))
print(flowCostsForAllClusters)
print(minFlowCosts)
```

```
[58.4378162363866, 20.998274926753876, 6.44676484190011, 5.296065588760511, 0.0, 0]
{4}
```