SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM

TNC TRANSACTIONS ON NETWORKS AND COMMUNICATIONS

VOLUME 6, NO. 2
ISSN: 2054 -7420

A Simple Greedy Algorithm for Energy-Efficient Communication in Small Multi-Interface Wireless Networks

[1,2]Christos Kaklamanis, [1]Stavros Maras, [1,2]Evi Papaioannou
[1]University of Patras, Patras, Greece;
[2]CTI "Diophantus", Patras, Greece;
kakl@ceid.upatras.gr; maras@ceid.upatras.gr; papaioan@ceid.upatras.gr

ABSTRACT

Wireless networks have become extremely popular recently due to the wide range of applications they support and also because sophisticated and affordable wireless devices like smartphones, tablets, etc have actually become part of our everyday life. Wireless devices have heterogeneous characteristics, like computational power, energy-consumption levels, supported communication protocols. Modern wireless devices are usually equipped with multiple radio interfaces like, WiFi, GPRS, Bluetooth, and can switch between different communication networks for meeting connectivity requirements and, thus improving quality of service and data collection perspectives. Establishing a connection between any two such devices requires that they are close and share at least one common available interface. If communication is established between two wireless devices, then the involved communication cost reflects the energy these devices consume and equals the cost for activating a particular common interface. In this setting, the objective is to suggest a cost-efficient interface activation plan which can guarantee low-cost communication for any two such wireless devices.

We model this practical problem as an instance of the Spanning Tree problem in an appropriately defined multigraph corresponding to the actual multi-interface wireless network. When connectivity is feasible, we propose and experimentally evaluate a simple greedy algorithm indicating which interfaces must be activated so that cost-efficient connectivity is established between any two wireless devices in the network.

Keywords: Small multi-interface wireless networks; Energy-efficient communication; Connectivity; Greedy algorithm.

1 Introduction

Wireless networks have become extremely popular during the recent years mainly due to the wide range of applications they support and also due to the fact that sophisticated and affordable wireless devices like smartphones, tablets, etc have actually become part of our everyday life. Wireless networks can be stand-alone network components (like for example a wireless network for a class or lab) or parts of larger networks and the Internet. Wireless devices have heterogeneous characteristics: they have different computational power, their energy-consumption levels vary, they support different communication protocols. Modern wireless devices are usually equipped with multiple radio interfaces like, for example,

DOI: 10.14738/tnc.62.4500
Publication Date: 1st May 2018
URL: http://dx.doi.org/10.14738/tnc.62.4500

WiFi, GPRS, Bluetooth. Therefore, they can switch between different communication networks for meeting connectivity requirements and improving quality of service. However, determining which interface must be activated on a wireless device depends on technical specifications of the device, communication requirements, connectivity constraints, necessary energy consumption. Even in the case that all other factors are neutral, energy consumption plays a crucial role for the selection of the interface to be activated, since, as long as a device runs out of battery, it can no longer be part of a wireless network. Besides benefits related to network infrastructure also data collection perspectives can be efficiently supported via the use of appropriate interfeces in multi-interface wireless networks.

We study a communication problem arising in wireless networks supporting multiple interfaces. These networks are composed of nodes which are wireless devices supporting some wireless interfaces. Communication between two such nodes requires the existence of at least one common interface and spatial proximity so that this specific shared interface can support their communication. If these conditions hold, then communication can be established. The involved cost essentially reflects the energy consumed and equals the cost of activating a particular interface which both nodes share. The objective is to activate interfaces at network nodes so that some connectivity property is maintained and the total activation cost is minimized. Various communication problems arise in multi-interface wireless networks based on the required connectivity property. We consider the problem termed as ConMI in [1] or Connectivity in [9]. In particular, we require that communication is established among all network nodes. The energy consumed by each device for the activation of a specific interface may vary substantially. Therefore, two cases are distinguished according to the interface activation costs: the more general one is when the activation cost for some interface is not the same at all network nodes; this is the heterogeneous case. In the homogeneous [1] or uniform cost [9] case, the cost of activating a particular interface is the same at all network nodes. Another important variant to some of the problems faced in multi-interface wireless networks concerns the total number of available interfaces supported in the overall network. The corresponding problems are in bounded or unbounded form depending on whether the total number of available interfaces is provided as a fixed constant or part of the input, respectively [9]. The unbounded version of such problems can be particularly useful for analytical results while the bounded version is more representative of practical cases.

Previous relevant work

Recent technological advances and a wide range of supported applications have made multi-interface wireless networks a very popular and wide-spread communication infrastructure. The study of communication problems arising in multi-interface wireless networks has attracted the interest of the research community. The key idea is to exploit the heterogeneity of the interfaces available in modern devices for reducing energy consumption and, consequently, extending network lifetime. Several well-known combinatorial optimization problems are then reconsidered with respect to this new feature.

Several basic problems studied for “traditional” wired and wireless networks have been reconsidered in this new setting [2], with an emphasis on problems related to network connectivity [3, 5] and routing [4]. Requirements for efficiency in energy consumption increase the complexity of these problems and raise new challenges. In [6], cost minimization in multi-interface wireless networks was studied. More precisely, given a graph representing desired connections between network nodes, the objective is to establish all graph edges by activating interfaces at network nodes of a minimum total cost. Several

variations of the problem are considered depending on the topology of the input graph (e.g., complete graphs, trees, planar graphs, bounded-degree graphs, general graphs) and on whether the number of interfaces is part of the input or a fixed constant. [6] considers both unit-cost interfaces and more general homogeneous instances. ConMI has been introduced in [7] which studies homogeneous instances of the problem. ConMI is proved to be APX-hard even when the graph modeling the network has a very special structure and the number of available interfaces is small (e.g., 2). In [7], a 2-approximation algorithm is presented by exploiting the relation of ConMI on homogeneous instances with the minimum spanning tree on an appropriately defined edge-weighted graph. Furthermore, [1] suggests an improved (3/2+ε)-approximation algorithm for ConMI. The algorithm is based on a challenging technique [10] that makes use of an "almost" minimum spanning tree in an appropriately defined hypergraph and transforms it to an efficient solution for connectivity. Better approximation bounds are obtained for special cases of ConMI such as the case of unit-cost interfaces. [9] provides a comprehensive survey on results from the recent relevant literature.

In this work, we focus on the heterogeneous, bounded form of ConMI. We model this practical problem as an instance of finding a spanning tree in an appropriately defined multigraph corresponding to the actual multi-interface wireless network. When connectivity is feasible, we propose, analyze and experimentally evaluate a simple greedy algorithm which indicates which interfaces must be activated so that cost-efficient connectivity is established between any two wireless devices in the network. We embed this technique into a proof-of-concept application for establishing energy-efficient communication within small groups of users (in classrooms, labs, meeting rooms, game rooms, etc) equipped with wireless devices (e.g., smartphones or tablets) supporting multiple interfaces. From a practical point of view, network infrastructure and data collection perspectives can highly benefit from the efficient management of available interfaces in multi-interface wireless networks.

The rest of the paper is structured as follows. In Section 2, we provide technical details regarding definitions and notation. In Section 3, we discuss our greedy approach to the connectivity problem in multi-interface wireless networks. We present experimental finding in Section 4 and conclude our report in Section 5.

2 Preliminaries: Definitions and Notation

In general, a multi-interface wireless network is modelled by a graph G = (V, E), where V represents the set of devices composing the network and E is the set of possible connections defined according to the distance between devices and the available interfaces that they share. Each $v \in V$ is associated with a set of available interfaces W(v). The set of all the possible available interfaces in the network is then determined by $\cup_{v \in v} W(v)$; we denote by k the cardinality of this set.

We say that a connection is established when the endpoint of the corresponding edge share at least one active interface. So, in our model, an edge $e_s = (u,v)_s$ exists for every interface s both u and v share, yielding a multigraph G which is assumed to be undirected and connected. If an interface s is activated at some node u, then u consumes some energy $c_u(s)$ for keeping s active and obtains a maximum communication bandwidth $b_u(s)$ with all its neighbors that share interface s. Furthermore, each possible edge $(u,v)_s$ has a cost equal to $c_u(s)+c_v(s)$.

For globally characterizing the interfaces each device supports, we use an interface assignment function W which covers graph G = (V, E), i.e., for each $(u,v) \in E$ it holds $W(u) \cap W(v) \neq \varnothing$. Our objective is to activate

interfaces at the nodes of V so that the resulting graph G' is a spanning tree for G, i.e. G' is connected, acyclic and spans all nodes of V. In the case when the resulting spanning tree G' has a minimum total edge-weight, we have a Minimum Spanning Tree (MST) for G.

Two classical deterministic greedy algorithms for constructing Minimum Spanning Trees are due to Kruskal [8] and Prim [11]. Both algorithms proceed by successively adding edges of smallest weight from those edges with a specified property that have not already been used. The main difference is the criterion used to select the next edge or edges to be added in each step. They are particularly simple and in fact solve the same problem by applying the greedy approach in two different ways and both always yield an optimal solution.

Kruskal's algorithm starts with an edge in the graph with minimum weight and builds the spanning tree by successively adding edges one by one into a growing spanning tree. It processes the edges in order of their weight values, from smallest to largest, including into the growing MST each edge which does not form a cycle with edges previously added. It stops after $|V|-1$ edges have been added. Kruskal's algorithm computes the MST of any connected edge-weighted graph with E edges and V vertices in time proportional to $|E|\log|E|$ (in the worst case) since sorting is the most time consuming operation.

Prim's algorithm constructs a minimum spanning tree incrementally, in a step-by-step fashion via a sequence of expanding subtrees. The initial subtree of the sequence consists of a single vertex selected arbitrarily from the set V of the vertices of the given graph. In each successive step, the algorithm expands the current tree greedily by simply adding to it the nearest vertex not in the tree. The distance of such a vertex is determined by the weight of the edge connecting it to the tree. In the case of at least two candidate nearest vertices, ties can be broken arbitrarily. The algorithm terminates when all vertices of the graph have been included in the spanning tree. Since the algorithm expands a tree by exactly one vertex during each step, the total number of required steps is n-1, where n is the number of vertices of V. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.

More precisely, the algorithm maintains two disjoint sets of vertices: one containing vertices that are in the growing spanning tree and another containing vertices not in the growing spanning tree. Then, it selects the lowest-cost vertex which is connected to the growing spanning tree but is not in the growing spanning tree and inserts it into the growing spanning tree. In order to avoid the creation of cycles, the algorithm marks the vertices which have been already selected and considers only those vertices that are not marked. Since each vertex is considered only once, the time complexity of the Prim's algorithm is $O((|V|+|E|)\log|V|)$.

3 Algorithm G_{MU}: A Simple Greedy Heuristic for Connectivity

3.1 Description and analysis

G_{MU} is a deterministic, greedy algorithm for connectivity in multi-interface wireless networks. It receives as input a graph G = (V,E) corresponding to a wireless network whose nodes support multiple interfaces. For each vertex $v \in V$ (representing a network node), information regarding supported interfaces and corresponding activation cost is provided. Each edge $e_s \in E$ connects pairs (u,v) of distinct vertices of V sharing interface s and has an associated weight equal to the sum of the activation cost of interface s at nodes u and v. Multiple edges are allowed between pairs of nodes sharing multiple (i.e., more than one)

interfaces. If $W(u) \cap W(v) \neq \varnothing$ for all $(u,v) \in E$, the algorithm returns a spanning tree $T = (V_T, E_T)$ for G, that is $V_T=V_G$, $E_T \subseteq E_G$.

More precisely, our algorithm works as follows. For a given input graph G = (V, E), the algorithm first checks whether there exists $(u,v) \in E$ for which the condition $W(u) \cap W(v) \neq \varnothing$ is false. If so, the algorithm terminates and fails to compute a spanning tree for G. If $W(u) \cap W(v) \neq \varnothing$ is true for all $(u,v) \in E$, a vertex set V_T is created and an arbitrary vertex $v \in V_G$ is added to it. Furthermore, a list L of edges is created where all edges $(u,v) \in E$ with $u \in V_T$ and $v \in V\backslash V_T$ are added. In this way, the formation of cycles in V_T is avoided. L is sorted in ascending order in terms of edge-weights. Then, the main loop of the algorithm is repeated until $V_T=V_G$. At each round, the first element of L (i.e., the edge of L of minimum weight) is added to T and L is updated so as to only include edges whose one endpoint belongs to V_T and the other is strictly in $V\backslash V_T$. Eventually, the algorithm terminates and returns a spanning tree T for G.

Below, the pseudocode for our greedy approach is presented.

Algorithm G_{MU}

Input: connected multigraph $G = (V_G, E_G)$

Output: spanning tree $T = (V_T, E_T)$, $V_T=V_G$, $E_T \subseteq E_G$

1. if there exists $(u,v) \in E$ for which $W(u) \cap W(v) = \varnothing$ then FAIL & TERMINATE; otherwise

2. V_T := a vertex of V_G chosen uniformly at random

3. L := all edges $(u,v) \in E_G$ such that $u \in V_T$ and $v \in V\backslash V_T$

4. While $V_T \neq V_G$

5. T := T $\cup$ min[L]

6. Update L (add/remove elements, sort)

7. RETURN spanning tree T & TERMINATE

Lemma 1 (Correctness)

Algorithm $\mathbf{G_{MU}}$ produces a spanning tree T for G=(V,E) when $W(u) \cap W(v) \neq \varnothing$, $\forall\ u,v \in V$.

Proof

If there exists $(u,v) \in E$ for which $W(u) \cap W(v) = \varnothing$, the algorithm terminates and fails to compute a spanning tree T for G (Step 1). Otherwise, a vertex $v \in V_G$ is chosen uniformly at random and added to V_T (Step 2). A list L is created containing all edges $(u,v) \in E_G$ such that $u \in V_T$ and $v \in V\backslash V_T$ (Step 3). If $|V_G|=1$, a (minimum) spanning tree T is returned with $|V_T|=|V_G|=1$ and the algorithm terminates (Step 7). Otherwise, the while loop is executed (Step 4).

In order to show that the returned graph is indeed a tree, we have to show that the resulted graph is connected and acyclic. Consider an instance after a while loop is executed and let $\{v_1, v_2, \ldots, v_n\} \in V_T$, $n < |V_G|$ and $\{v_{n+1}, \ldots, v_{|VG|}\} \in V_G\backslash V_T$. Assume that $e=(v_i, v_j)$ is an edge currently considered for addition to T. This implies that one of the endpoints of e, say v_i, must be in V_T. Then, for a cycle to be created, v_j must be

a vertex already visited, i.e., a vertex also in V_T. This is a contradiction since every edge $e=(v_i, v_j)$ considered for addition to T must have $v_i \in V_T$ and $v_j \in V \backslash V_T$ (or vice-versa). Furthermore, T will eventually contain all nodes of G, since T is gradually augmented until $V_T=V_G$. This completes the proof of Lemma 1.

Lemma 2 (Time complexity)

Algorithm $\mathbf{G_{MU}}$ requires $O(|V^3|)$ steps.

Proof

Assuming that the number of available interfaces is $O(|V|)$, Step 1 requires $O(|V^3|)$ steps, since $O(|V^2|)$ pairs have to be checked. Furthermore, $O(|V|)$ repetitions of the while loop (Step 4) are required. Adding an edge to the list L (Step 5) requires $O(|V|^2)$ steps. Updating L requires $O(|V||E|)$ steps and sorting L requires $O(|V||E|\log|E|)$ steps (Step 6). This gives an overall time complexity of $O(|V|^3)$.

Lemma 3 (Activation cost)

Algorithm $\mathbf{G_{MU}}$ yields a total activation cost of O(V).

Proof

The spanning tree computed by algorithm G_{MU} for an input graph G=(V,E) representing a wireless network whose nodes support multiple interfaces has |V-1| edges. Assuming that c is the maximum activation cost taken over all available interfaces yields a maximum total cost of O(V).

3.2 Implementation

Software and hardware

For our experimental study, we used Python 3. We preferred Python to other popular programming languages, like C++ or Java, because it is friendly to use and easy to learn; yet, it is a powerful programming language which allows simple and flexible representations of networks as well as clear and concise expressions of network algorithms. Python can be used on many operating systems, providing a standard library and plenty of community-contributed modules. Python is developed under an open source license, making it freely usable and distributable [13]. Python 3, in particular, offers new important programming features and facilities as well as improved memory management.

Furthermore, we used NetworkX for our implementation. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. NetworkX provides improved features regarding numerical linear algebra and drawing and can facilitate tasks including loading and storing networks in various data formats, generation of random and classic networks, analysis of network structure, building network models, drawing networks, and so on. NetworkX is freely usable and distributable under the terms of the 3-clause BSD License [12].

We implemented and executed our experiments on a mac OSX machine with an Intel Core i7 2,2 GHz processor and 16 GB 1333 MHz DDR3 RAM.

Input

The input multigraph corresponding to a wireless network supporting multiple interfaces can be provided to our code either manually or automatically.

Providing the input manually requires the use of NetworkX packages together with a basic program in Python. First, network nodes are defined. For each node a unique id and the interfaces it supports together with their activation cost must be given. Then, connections between nodes are defined in terms of graph edges. Multiple edges are allowed between each pair of network nodes; each edge corresponds to an interface shared by the nodes of the pair. For each edge, its endpoints and its weight must be given. The weight of an edge (u,v) corresponding to a shared interface s equals the sum of the activation cost of interface s at nodes u and v. Figure 1 shows an input instance provided manually.

```
G.add_node(1, Interface1 = "yes", Interface2 = "no", ActCost1 = 10)
G.add_node(2, Interface1 = "yes", Interface2 = "yes", ActCost1 = 5, ActCost2 = 6)
G.add_node(3, Interface1 = "yes", Interface2 = "yes", ActCost1 = 22, ActCost2 = 20)
G.add_node(4, Interface1 = "yes", Interface2 = "yes", ActCost1 = 19, ActCost2 = 10)
G.add_node(5, Interface1 = "yes", Interface2 = "no", ActCost1 = 13)
G.add_node(6, Interface1 = "yes", Interface2 = "yes", ActCost1 = 12, ActCost2 = 10)
G.add_edge(1,2,key=1,weight=15)
G.add_edge(1,3,key=1,weight=32)
G.add_edge(1,4,key=1,weight=29)
G.add_edge(1,5,key=1,weight=23)
G.add_edge(1,6,key=1,weight=22)
G.add_edge(2,3,key=1,weight=27)
G.add_edge(2,3,key=2,weight=26)
G.add_edge(2,4,key=1,weight=24)
G.add_edge(2,4,key=2,weight=16)
G.add_edge(2,5,key=1,weight=18)
G.add_edge(2,6,key=1,weight=17)
G.add_edge(2,6,key=2,weight=16)
G.add_edge(3,4,key=1,weight=41)
G.add_edge(3,4,key=2,weight=30)
G.add_edge(3,5,key=1,weight=35)
G.add_edge(3,6,key=1,weight=34)
G.add_edge(4,5,key=1,weight=32)
G.add_edge(4,6,key=2,weight=20)
G.add_edge(5,6,key=1,weight=25)
```

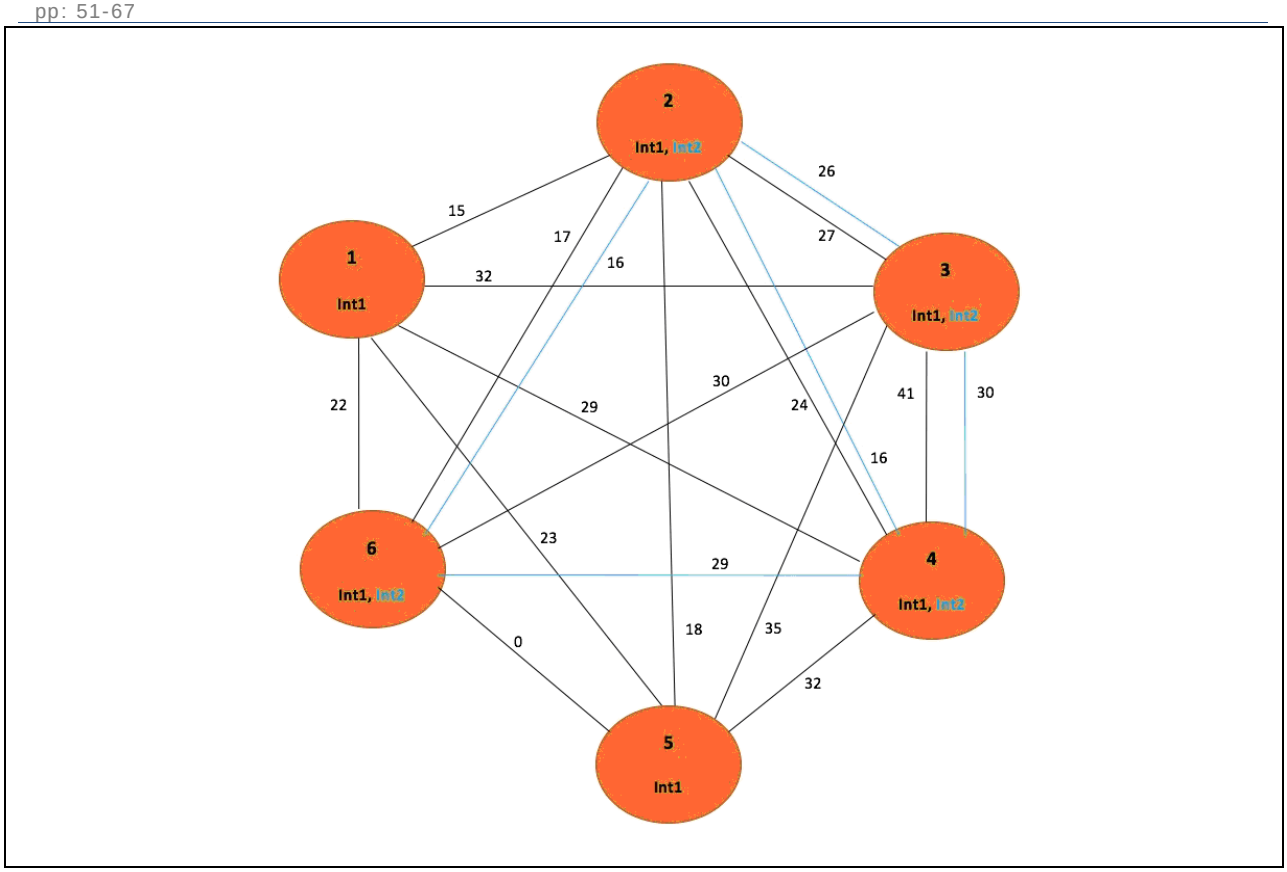

Figure 1: Manual generation of an input instance.

Providing the input automatically requires the execution of the "graph generator" function depicted in Figure 2.

```
# Graph generator function
#
# create the Vertices
#
for n in range(numOfVertices):
        n_corr = n+1
        G.add_node(n_corr)
        # randomly decide if Interface1 and Interface2 exists. If exists, add also an activation cost
        if random.randint(0,30)>0:
                G.node[n_corr]['Interface1'] = 'yes'
                G.node[n_corr]['ActCost1'] = random.randint(10,100)
        else:
                G.node[n_corr]['Interface1'] = 'no'
        if random.randint(0,30)>0:
                G.node[n_corr]['Interface2'] = 'yes'
                G.node[n_corr]['ActCost2'] = random.randint(10,100)
        else:
                G.node[n_corr]['Interface2'] = 'no'
for inj in G.nodes(data=True):
        if (inj[1]['Interface1'] == 'no') & (inj[1]['Interface2'] == 'no'):
                print('Not every pair of nodes have at least one common interface. Algorithm CANNOT execute')
                sys.exit()
#
# create the Edges
#
```

```
# add edges with Interface 1 in common
for ne in range(numOfEdges):
        n1 = random.randint(1,numOfVertices)
        n2 = random.randint(1,numOfVertices)
        # make sure the 2 randomly selected vertices are not the same
        if n1 == n2:
                n2 =+ 1
        if (G.node[n1]['Interface1'] == 'yes') & (G.node[n2]['Interface1'] == 'yes'):
                edgeWeight = G.node[n1]['ActCost1'] + G.node[n2]['ActCost1']
                G.add_edge(n1,n2,key=1,weight=edgeWeight)
# add edges with Interface 2 in common
for ne in range(numOfEdges):
        n1 = random.randint(1,numOfVertices)
        n2 = random.randint(1,numOfVertices)
        # make sure the 2 randomly selected vertices are not the same
        if n1 == n2:
                n2 =+ 1
        if (G.node[n1]['Interface2'] == 'yes') & (G.node[n2]['Interface2'] == 'yes'):
                edgeWeight = G.node[n1]['ActCost2'] + G.node[n2]['ActCost2']
                G.add_edge(n1,n2,key=2,weight=edgeWeight)
# print nodes and edges of graph
for g in G.nodes(data=True):
        print(g)
for i in G.edges(data=True, keys=True):
        print(i)
start=time.time() # start the timing of the algorithm
# check if every pair of nodes, connected with an edge, have at least one common interface
for itedge in G.edges(data=True):
        if not ((G.node[itedge[0]]['Interface1'] == 'yes') & (G.node[itedge[0]]['Interface1'] == 'yes')) \
        | ((G.node[itedge[0]]['Interface2'] == 'yes') & (G.node[itedge[0]]['Interface2'] == 'yes')):
                print('Not every pair of nodes have at least one common interface. Algorithm CANNOT execute')
                sys.exit()
```

Figure 2: "graph generator" function.

Our "graph generator" works as follows. Initially, the total number of graph vertices is randomly chosen via the use of function "random". Then, the input multigraph is generated by 4 consecutive "for" loops. The first "for" loop generates the vertices of the graph (attributing to each of them an id, supported interfaces and interface activation costs). The second "for" loop verifies that there exists at least one shared interface between each pair of vertices; if this is not the case, the "graph generator" terminates and restarts. The last two "for" loops are then used to generate the edges of the multigraph. The total number of edges is generated via the use of function "random". Endpoints are also randomly assigned to edges. Then, a verification process checks for edges having both endpoints assigned the same vertex. If no such edge exists, edge endpoints are checked for shared interfaces and the final input graph is produced.

Main part of the code

The core component of our code is presented in Figure 3, implements algorithm G_{MU} and works as follows. Initially, an arbitrary graph vertex is selected, assigned to variable ranV and printed on the screen. Then, a list L is created containing already explored vertices; ranV is inserted to L. Furthermore, a second list, namely sortEdges, is created containing edges to be considered for addition to the generated spanning tree; these are edges whose one endpoint is a vertex already explored (and, therefore, included in list L) and their other endpoint is a vertex not yet explored. The list sortEdges is sorted in ascending order of weight of included edges. The required spanning tree is then built through a main "while" loop. In particular, elements of the list sortEdges are printed on the screen, the edge e_{min} of the minimum weight

is assigned to variable ST (which corresponds to the spanning tree under generation) and the just explored endpoint of e_{min} is also added to L. Then, the list sortEdges is updated.

```
#
# st on conMI
#
# randomly select a vertex
ranV = random.randint(1,G.number_of_nodes())
print('Begin algorithm with randomly selected vertex:', ranV)
# create list of visited nodes and append the randomly selected vertex
l = []
l.append(ranV)
# find edge with minimum weight
sortEdges = sorted(G.edges([ranV],data='weight',keys=True),key=itemgetter(3))
while list(G.nodes()) != sorted(l):
        # print avalailable edges, select and prin the one with minimum weight
        print('edges to choose from:' ,sortEdges)
        print('edge in process',sortEdges[0])
        # add the selected edge to the spanning tree and its node to the list of already visited nodes
        ST.add_edge(sortEdges[0][0],sortEdges[0][1],weight=sortEdges[0][3],activatedInterface=sortEdges[0][2])
        print('Node to enter list:',sortEdges[0][1])
        l.append(sortEdges[0][1])
        # add the extra edges, based on the updated visited nodes' list
        sortEdges = sorted(G.edges([*l],data='weight',keys=True),key=itemgetter(3))
        # remove edges that both of vertices already exist on spanning tree
        for it in sortEdges:
                if not((it[0] in l) & (it[1] in l)):
                        filtSortEdges.append(it)
        sortEdges = filtSortEdges
        filtSortEdges = []
        l = list(set(l))
        print('visited nodes so far', l)
        print('------------------------------\n')
end = time.time() # finish the timing of the algorithm
# print the spanning tree
print('\nSpanning tree:')
for a,b,c in ST.edges(data=True):
        print('Edge',a,b,'has',c)
print("\nTIME:",end-start)
# initializing already activated attribute for ST nodes
for n in ST.nodes():
        ST.node[n]['alrAct1'] = 0
        ST.node[n]['alrAct2'] = 0
# spanning tree cost
sp = 0
for ed in ST.edges(data=True):
        if (ed[2]['activatedInterface'] == 1) :
                if (ST.node[ed[0]]['alrAct1']!=1) & (ST.node[ed[1]]['alrAct1']!=1) :
                        sp += ed[2]['weight']
                        ST.node[ed[0]]['alrAct1']=1
                        ST.node[ed[1]]['alrAct1']=1
                elif (ST.node[ed[0]]['alrAct1']!=1):
                        sp += G.node[ed[0]]['ActCost1']
                        ST.node[ed[0]]['alrAct1']=1
                elif (ST.node[ed[1]]['alrAct1']!=1):
                        sp += G.node[ed[1]]['ActCost1']
                        ST.node[ed[1]]['alrAct1']=1
        else :
                if (ST.node[ed[0]]['alrAct2']!=1) & (ST.node[ed[1]]['alrAct2']!=1) :
```

```
                                sp += ed[2]['weight']
                                ST.node[ed[0]]['alrAct2']=1
                                ST.node[ed[1]]['alrAct2']=1
                        elif (ST.node[ed[0]]['alrAct2']!=1):
                                sp += G.node[ed[0]]['ActCost2']
                                ST.node[ed[0]]['alrAct2']=1
                        elif (ST.node[ed[1]]['alrAct2']!=1):
                                sp += G.node[ed[1]]['ActCost2']
                                ST.node[ed[1]]['alrAct2']=1
#  cost for activating all available interfaces
costAll = 0
for ed in G.nodes(data='ActCost1'):
        if (ed[1]):
                costAll += ed[1]
for ed in G.nodes(data='ActCost2'):
        if (ed[1]):
                costAll += ed[1]
print('\nSpanning Tree cost',sp,'and Cost for activating all available interfaces',costAll,'\n')
```

Figure 3: The program implementing G_{MU}.

Figure 4(b) shows how our code works when executed on the input multigraph of Figure 4(a).

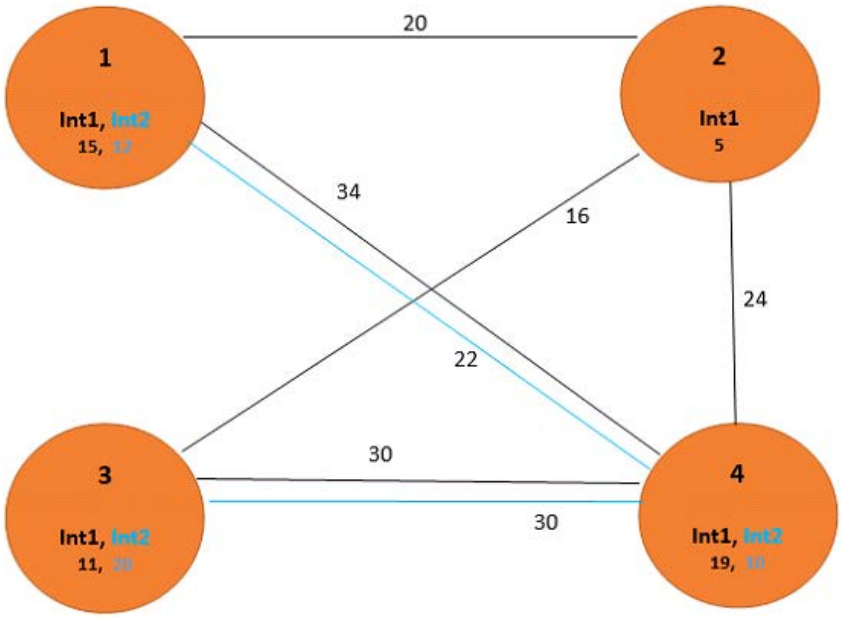

Figure 4(a): input multigraph.

```
Begin algorithm with randomly selected vertex: 3
edges to choose from: [(3, 2, 1, 16), (3, 4, 1, 30), (3, 4, 2, 30)]
edge in process (3, 2, 1, 16)
Node to enter list: 2
visited nodes so far [2, 3]
------------------------------

edges to choose from: [(2, 1, 1, 20), (2, 4, 1, 24), (3, 4, 1, 30), (3, 4, 2, 30)]
edge in process (2, 1, 1, 20)
Node to enter list: 1
visited nodes so far [1, 2, 3]
------------------------------

edges to choose from: [(1, 4, 2, 22), (2, 4, 1, 24), (3, 4, 1, 30), (3, 4, 2, 30), (1, 4, 1, 34)]
edge in process (1, 4, 2, 22)
Node to enter list: 4
visited nodes so far [1, 2, 3, 4]
------------------------------


Spanning tree:
Edge 3 2 has {'weight': 16, 'activatedInterface': 1}
Edge 2 1 has {'weight': 20, 'activatedInterface': 1}
Edge 1 4 has {'weight': 22, 'activatedInterface': 2}

TIME: 0.0004992485046386719

Spanning Tree cost 53 and Cost for activating all available interfaces 92
```

Figure 4(b): Generation of a spanning tree for the multigraph of part (a) by G_{MU}.

Each run is followed by the list of edges of the generated spanning tree, the total time (in seconds) required for the spanning tree generation, to total activation cost and the total cost for activating all available interfaces at all network nodes.

4 Experimental results

For our experimental analysis, a total of 30 automatically generated input instances have been used. Assuming the existence of 2 available interfaces, algorithm G_{MU} has been used to compute spanning trees for these instances. Below, we first give an example of an automatically generated input instance; then, we provide charts for the performance of our approach in terms of activation cost and execution time.

4.1 An example of an automatically generated input instance

Below, we provide screen captures of an indicative experiment. A wireless network of 38 nodes is automatically generated; there are 2 available interfaces. Figure 5 shows network nodes. Edges are depicted in Figure 6.

```
(1, {'Interface1': 'yes', 'ActCost1': 41, 'Interface2': 'yes', 'ActCost2': 23})
(2, {'Interface1': 'yes', 'ActCost1': 59, 'Interface2': 'yes', 'ActCost2': 18})
(3, {'Interface1': 'yes', 'ActCost1': 54, 'Interface2': 'yes', 'ActCost2': 12})
(4, {'Interface1': 'yes', 'ActCost1': 90, 'Interface2': 'yes', 'ActCost2': 99})
(5, {'Interface1': 'yes', 'ActCost1': 21, 'Interface2': 'yes', 'ActCost2': 36})
(6, {'Interface1': 'yes', 'ActCost1': 88, 'Interface2': 'yes', 'ActCost2': 82})
(7, {'Interface1': 'yes', 'ActCost1': 36, 'Interface2': 'yes', 'ActCost2': 82})
(8, {'Interface1': 'yes', 'ActCost1': 21, 'Interface2': 'yes', 'ActCost2': 100})
(9, {'Interface1': 'yes', 'ActCost1': 68, 'Interface2': 'yes', 'ActCost2': 68})
(10, {'Interface1': 'yes', 'ActCost1': 86, 'Interface2': 'yes', 'ActCost2': 58})
(11, {'Interface1': 'yes', 'ActCost1': 36, 'Interface2': 'yes', 'ActCost2': 69})
(12, {'Interface1': 'yes', 'ActCost1': 96, 'Interface2': 'yes', 'ActCost2': 70})
(13, {'Interface1': 'yes', 'ActCost1': 70, 'Interface2': 'yes', 'ActCost2': 78})
(14, {'Interface1': 'yes', 'ActCost1': 82, 'Interface2': 'yes', 'ActCost2': 27})
(15, {'Interface1': 'yes', 'ActCost1': 38, 'Interface2': 'yes', 'ActCost2': 14})
(16, {'Interface1': 'yes', 'ActCost1': 55, 'Interface2': 'yes', 'ActCost2': 40})
(17, {'Interface1': 'yes', 'ActCost1': 28, 'Interface2': 'yes', 'ActCost2': 80})
(18, {'Interface1': 'yes', 'ActCost1': 57, 'Interface2': 'yes', 'ActCost2': 76})
(19, {'Interface1': 'yes', 'ActCost1': 21, 'Interface2': 'yes', 'ActCost2': 21})
(20, {'Interface1': 'yes', 'ActCost1': 24, 'Interface2': 'yes', 'ActCost2': 98})
(21, {'Interface1': 'yes', 'ActCost1': 56, 'Interface2': 'yes', 'ActCost2': 96})
(22, {'Interface1': 'yes', 'ActCost1': 24, 'Interface2': 'yes', 'ActCost2': 31})
(23, {'Interface1': 'yes', 'ActCost1': 73, 'Interface2': 'yes', 'ActCost2': 36})
(24, {'Interface1': 'yes', 'ActCost1': 27, 'Interface2': 'yes', 'ActCost2': 32})
(25, {'Interface1': 'yes', 'ActCost1': 27, 'Interface2': 'yes', 'ActCost2': 27})
(26, {'Interface1': 'yes', 'ActCost1': 13, 'Interface2': 'no'})
(27, {'Interface1': 'yes', 'ActCost1': 25, 'Interface2': 'no'})
(28, {'Interface1': 'yes', 'ActCost1': 66, 'Interface2': 'yes', 'ActCost2': 92})
(29, {'Interface1': 'yes', 'ActCost1': 15, 'Interface2': 'yes', 'ActCost2': 28})
(30, {'Interface1': 'yes', 'ActCost1': 36, 'Interface2': 'yes', 'ActCost2': 58})
(31, {'Interface1': 'yes', 'ActCost1': 67, 'Interface2': 'yes', 'ActCost2': 81})
(32, {'Interface1': 'no', 'Interface2': 'yes', 'ActCost2': 92})
(33, {'Interface1': 'yes', 'ActCost1': 34, 'Interface2': 'yes', 'ActCost2': 72})
```

Figure 5: Nodes of an automatically generated input instance with 2 available interfaces

```
(17, 29, 2, {'weight': 108})
(17, 24, 2, {'weight': 112})
(17, 28, 2, {'weight': 172})
(17, 33, 2, {'weight': 152})
(18, 24, 1, {'weight': 84})
(18, 32, 2, {'weight': 168})
(19, 29, 1, {'weight': 36})
(19, 20, 1, {'weight': 45})
(19, 23, 1, {'weight': 94})
(19, 33, 1, {'weight': 55})
(19, 24, 2, {'weight': 53})
(20, 21, 1, {'weight': 80})
(20, 23, 1, {'weight': 97})
(21, 23, 1, {'weight': 129})
(21, 22, 1, {'weight': 80})
(21, 30, 2, {'weight': 154})
(22, 31, 1, {'weight': 91})
(22, 29, 2, {'weight': 59})
(23, 30, 1, {'weight': 109})
(23, 33, 2, {'weight': 108})
(24, 31, 2, {'weight': 113})
(24, 25, 2, {'weight': 59})
(25, 30, 1, {'weight': 63})
(25, 31, 1, {'weight': 94})
(25, 31, 2, {'weight': 108})
(25, 27, 1, {'weight': 52})
(25, 26, 1, {'weight': 40})
(25, 28, 2, {'weight': 119})
(26, 28, 1, {'weight': 79})
(28, 30, 1, {'weight': 102})
(28, 30, 2, {'weight': 150})
(28, 32, 2, {'weight': 184})
(29, 31, 1, {'weight': 82})
(30, 31, 1, {'weight': 103})
(30, 31, 2, {'weight': 139})
(30, 33, 2, {'weight': 130})
(32, 33, 2, {'weight': 164})
```

```
(1, 31, 1, {'weight': 108})
(1, 31, 2, {'weight': 104})
(1, 11, 1, {'weight': 77})
(1, 23, 1, {'weight': 114})
(1, 22, 1, {'weight': 65})
(1, 22, 2, {'weight': 54})
(1, 16, 1, {'weight': 96})
(1, 12, 1, {'weight': 137})
(1, 12, 2, {'weight': 93})
(1, 13, 1, {'weight': 111})
(1, 5, 1, {'weight': 62})
(1, 24, 1, {'weight': 68})
(1, 21, 2, {'weight': 119})
(1, 10, 2, {'weight': 81})
(1, 8, 2, {'weight': 123})
(1, 25, 2, {'weight': 50})
(1, 18, 2, {'weight': 99})
(1, 33, 2, {'weight': 95})
(1, 2, 2, {'weight': 41})
(1, 9, 2, {'weight': 91})
(2, 27, 1, {'weight': 84})
(2, 11, 1, {'weight': 95})
(2, 11, 2, {'weight': 87})
(2, 13, 1, {'weight': 129})
(2, 4, 1, {'weight': 149})
(2, 12, 1, {'weight': 155})
(2, 8, 1, {'weight': 80})
(2, 30, 1, {'weight': 95})
(2, 3, 1, {'weight': 113})
(2, 6, 2, {'weight': 100})
(2, 32, 2, {'weight': 110})
(2, 28, 2, {'weight': 110})
(2, 19, 2, {'weight': 39})
(2, 9, 2, {'weight': 86})
(3, 29, 1, {'weight': 69})
(3, 13, 1, {'weight': 124})
(3, 7, 1, {'weight': 90})
```

```
(3, 7, 2, {'weight': 94})
(3, 11, 1, {'weight': 90})
(3, 17, 1, {'weight': 82})
(3, 20, 1, {'weight': 78})
(3, 14, 2, {'weight': 39})
(3, 30, 2, {'weight': 70})
(3, 5, 2, {'weight': 48})
(3, 22, 2, {'weight': 43})
(4, 17, 1, {'weight': 118})
(4, 12, 1, {'weight': 186})
(4, 30, 1, {'weight': 126})
(4, 30, 2, {'weight': 157})
(4, 21, 1, {'weight': 146})
(4, 8, 1, {'weight': 111})
(4, 16, 2, {'weight': 139})
(4, 24, 2, {'weight': 131})
(4, 28, 2, {'weight': 191})
(4, 32, 2, {'weight': 191})
(5, 24, 1, {'weight': 48})
(5, 19, 1, {'weight': 42})
(5, 29, 1, {'weight': 36})
(5, 28, 1, {'weight': 87})
(5, 21, 1, {'weight': 77})
(5, 20, 1, {'weight': 45})
(5, 6, 1, {'weight': 109})
(5, 9, 2, {'weight': 104})
(5, 8, 2, {'weight': 136})
(5, 10, 2, {'weight': 94})
(6, 10, 1, {'weight': 174})
(6, 10, 2, {'weight': 140})
(6, 9, 1, {'weight': 156})
(6, 15, 1, {'weight': 126})
(6, 29, 1, {'weight': 103})
(6, 14, 1, {'weight': 170})
(6, 28, 2, {'weight': 174})
(6, 25, 2, {'weight': 109})
(6, 30, 2, {'weight': 140})
```

```
(6, 33, 2, {'weight': 154})
(7, 11, 1, {'weight': 72})
(7, 16, 1, {'weight': 91})
(7, 21, 1, {'weight': 92})
(7, 14, 1, {'weight': 118})
(7, 14, 2, {'weight': 109})
(7, 26, 1, {'weight': 49})
(7, 31, 1, {'weight': 103})
(7, 20, 1, {'weight': 60})
(7, 8, 1, {'weight': 57})
(7, 29, 2, {'weight': 110})
(7, 15, 2, {'weight': 96})
(7, 13, 2, {'weight': 160})
(7, 17, 2, {'weight': 162})
(7, 32, 2, {'weight': 174})
(8, 20, 1, {'weight': 45})
(8, 13, 1, {'weight': 91})
(8, 15, 1, {'weight': 59})
(8, 15, 2, {'weight': 114})
(8, 11, 1, {'weight': 57})
(8, 24, 1, {'weight': 48})
(8, 29, 1, {'weight': 36})
(8, 28, 2, {'weight': 192})
(9, 31, 1, {'weight': 135})
(9, 17, 1, {'weight': 96})
(9, 18, 1, {'weight': 125})
(9, 22, 1, {'weight': 92})
(9, 22, 2, {'weight': 99})
(9, 13, 2, {'weight': 146})
(9, 15, 2, {'weight': 82})
(10, 19, 1, {'weight': 107})
(10, 12, 1, {'weight': 182})
(10, 22, 1, {'weight': 110})
(10, 26, 1, {'weight': 99})
(10, 21, 1, {'weight': 142})
(10, 32, 2, {'weight': 150})
(10, 13, 2, {'weight': 136})
(11, 25, 1, {'weight': 63})
(12, 16, 1, {'weight': 151})
(12, 33, 1, {'weight': 130})
(12, 23, 1, {'weight': 169})
(12, 28, 2, {'weight': 162})
(12, 19, 2, {'weight': 91})
(12, 15, 2, {'weight': 84})
(12, 25, 2, {'weight': 97})
(13, 17, 1, {'weight': 98})
(13, 21, 1, {'weight': 126})
(13, 26, 1, {'weight': 83})
(13, 18, 2, {'weight': 154})
(13, 30, 2, {'weight': 136})
(13, 20, 2, {'weight': 176})
(13, 33, 2, {'weight': 150})
(13, 24, 2, {'weight': 110})
(13, 15, 2, {'weight': 92})
(13, 29, 2, {'weight': 106})
(14, 22, 1, {'weight': 106})
(14, 18, 1, {'weight': 139})
(14, 18, 2, {'weight': 103})
(14, 15, 2, {'weight': 41})
(14, 17, 2, {'weight': 107})
(14, 24, 2, {'weight': 59})
(15, 23, 1, {'weight': 111})
(15, 20, 1, {'weight': 62})
(15, 16, 1, {'weight': 93})
(15, 33, 1, {'weight': 72})
(15, 22, 1, {'weight': 62})
(15, 22, 2, {'weight': 45})
(15, 17, 2, {'weight': 94})
(15, 31, 2, {'weight': 95})
(16, 30, 1, {'weight': 91})
(17, 22, 1, {'weight': 52})
(17, 20, 2, {'weight': 178})
(17, 31, 2, {'weight': 161})
(17, 32, 2, {'weight': 172})
```

Figure 6: Edges of an automatically generated input instance with 2 available interfaces

Algorithm GMU executed for the input instance presented above computed the spanning tree depicted in Figure 7.

```
Spanning tree:
Edge 20 8 has {'weight': 45, 'activatedInterface': 1}
Edge 8 29 has {'weight': 36, 'activatedInterface': 1}
Edge 8 11 has {'weight': 57, 'activatedInterface': 1}
Edge 8 4 has {'weight': 111, 'activatedInterface': 1}
Edge 29 19 has {'weight': 36, 'activatedInterface': 1}
Edge 29 5 has {'weight': 36, 'activatedInterface': 1}
Edge 29 31 has {'weight': 82, 'activatedInterface': 1}
Edge 19 2 has {'weight': 39, 'activatedInterface': 2}
Edge 19 33 has {'weight': 55, 'activatedInterface': 1}
Edge 19 23 has {'weight': 94, 'activatedInterface': 1}
Edge 5 24 has {'weight': 48, 'activatedInterface': 1}
Edge 5 3 has {'weight': 48, 'activatedInterface': 2}
Edge 5 21 has {'weight': 77, 'activatedInterface': 1}
Edge 2 1 has {'weight': 41, 'activatedInterface': 2}
Edge 2 6 has {'weight': 100, 'activatedInterface': 2}
Edge 2 32 has {'weight': 110, 'activatedInterface': 2}
Edge 1 25 has {'weight': 50, 'activatedInterface': 2}
Edge 1 10 has {'weight': 81, 'activatedInterface': 2}
Edge 24 18 has {'weight': 84, 'activatedInterface': 1}
Edge 3 14 has {'weight': 39, 'activatedInterface': 2}
Edge 3 22 has {'weight': 43, 'activatedInterface': 2}
Edge 14 15 has {'weight': 41, 'activatedInterface': 2}
Edge 15 9 has {'weight': 82, 'activatedInterface': 2}
Edge 15 12 has {'weight': 84, 'activatedInterface': 2}
Edge 22 17 has {'weight': 52, 'activatedInterface': 1}
Edge 25 26 has {'weight': 40, 'activatedInterface': 1}
Edge 25 27 has {'weight': 52, 'activatedInterface': 1}
Edge 25 30 has {'weight': 63, 'activatedInterface': 1}
Edge 26 7 has {'weight': 49, 'activatedInterface': 1}
Edge 26 28 has {'weight': 79, 'activatedInterface': 1}
Edge 26 13 has {'weight': 83, 'activatedInterface': 1}
Edge 7 16 has {'weight': 91, 'activatedInterface': 1}

TIME: 0.01951456069946289

Spanning Tree cost 1501 and Cost for activating all available interfaces 3330
```

Figure 7: The spanning tree computed by algorithm G_{MU} for the input instance of Figure 5.

4.2 Activation cost

Figure 8 shows how the network size affects the performance of our greedy approach in terms of total activation cost. Assuming that the input multigraph is connected, we compare the total activation cost induced by G_{MU} to the cost of activating all available interfaces at all network nodes and, also, to O(|V|).

In terms of activation cost, G_{MU} obtains a performance linear in the size of the network; this implies that an extremely simple greedy solution can offer a satisfactory performance as long as the network size remains limited. Indeed, small-scale wireless networks composed of devices supporting a small number of interfaces do appear often in school classes, labs, meeting rooms, medical councils, etc. In such cases, a simple greedy approach like G_{MU}, although theoretically deemed to perform much worse than significantly more complex approaches from the recent literature, can suggest a useful practical solution.

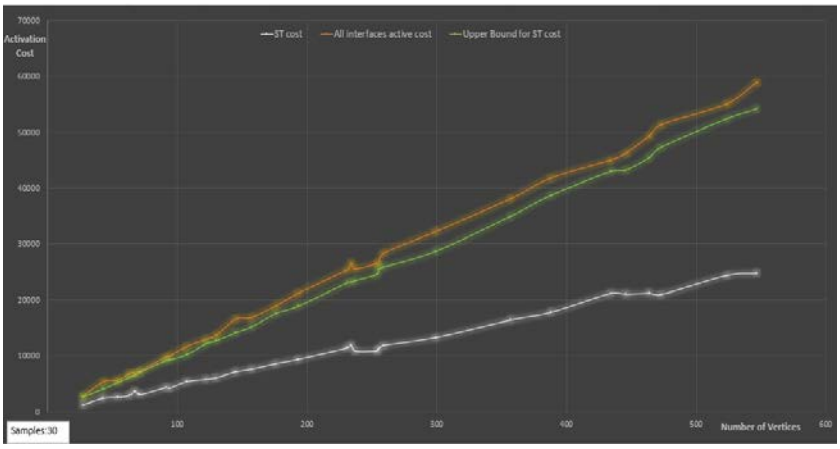

Figure 8: Activation cost of G_{MU} (white line) vs an upper bound for the ST activation cost (green line) and the cost for activating all available interfaces (orange line).

4.3 Execution time

Figure 9 shows how the network size affects the performance of our greedy approach in terms of execution time. Measurements were taken using the function “time” offered by Python. We measured the time interval needed to compute a spanning tree for an input graph G by algorithm G_{MU}. As it can be observed, for networks composed of at most 100 nodes, the running time of G_{MU} is less that 1 sec; for larger networks of approximately 500 nodes, the running time of G_{MU} remains low (at most 1 min) in practice.

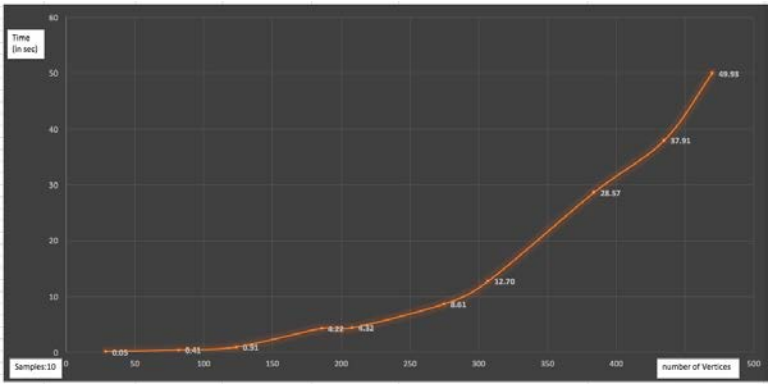

Figure 9: Running time of G_{MU}.

5 Conclusion

We addressed the problem of connectivity in small wireless networks composed of devices supporting multiple interfaces. From a practical point of view, network infrastructure and data collection perspectives can highly benefit from the efficient management of available interfaces in multi-interface wireless networks. We modelled this practical problem as an instance of the Spanning Tree problem in an appropriately defined multigraph corresponding to the actual multi-interface wireless network. We suggested a simple greedy algorithm that indicates which interfaces must be activated so that cost-efficient connectivity is established between any two wireless devices in the network. Our approach shows that simple solutions of theoretically poor performance can still be interesting in practice.

Our future plans include the implementation of the randomized polynomial-time approximation scheme of Prömel and Steger for solving almost exactly the MST problem in hypergraphs [10]; such an implementation would be extremely interesting as a stand-alone component but also as a building block of the (3/2+ε)-approximation algorithm presented in [1] for connectivity in multi-interface wireless networks.

REFERENCES

[1] Athanassopoulos, S., Caragiannis, I., Kaklamanis, C., Papaioannou, E., Energy-efficient communication in multi-interface wireless networks. Theory of Computing Systems, 2013. 52 (2), p. 285-296.

[2] Bahl, P., Adya, A., Padhye, J., Walman, A., Reconsidering wireless systems with multiple radios. ACM SIGCOMM Computer Communication Review, 2004. 34(5), p. 39-46.

[3] Cavalcanti, D., Gossain, H., Agrawal, D., Connectivity in multi-radio, multi-channel heterogeneous ad hoc networks. In Proceedings of the IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 05), 2005. IEEE Press, New York, p. 1322-1326.

[4] Draves, R., Padhye, J., Zill, B., Routing in multi-radio, multi-hop wireless mesh networks. In Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (Mobi- Com 04), 2004. ACM, New York, p. 114-128.

[5] Faragó, A., Basagni, S., The effect of multi-radio nodes on network connectivity: a graph theoretic analysis. In Proceedings of the IEEE 19th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 08), 2008. IEEE Press, New York.

[6] Klasing, R., Kosowski, A., Navarra, A., Cost minimisation in wireless networks with bounded and unbounded number of interfaces. Networks, 2009. 53(3), p. 266-275.

[7] Kosowski, A., Navarra, A., Pinotti, M.C., Exploiting multi-interface networks: connectivity and cheapest paths. Wireless Networks, 2010. 16(4), p. 1063-1073.

[8] Kruskal, J., On the shortest spanning subtree of a graph and the traveling salesman problem. In Proceedings of the American Mathematical Society, 1956. 7, p. 48-50.

[9] Navarra, A., D'Angelo, G., Di Stefano, G., Multi-interface wireless networks: complexity and algorithms. Chapter in Wireless Sensor Networks: From Theory to Applications, CRC Press, Taylor & Francis Group, 2014. p. 119-156.

[10] Prömel, H.J., Steger, A., A new approximation algorithm for the Steiner tree problem with performance ratio 5/3. Journal of Algorithms, 2000. 36, p. 89-101.

[11] Prim, R. C., Shortest connection networks and some generalizations. Bell System Technical Journal, 1957. 36(6), p. 1389-1401.

[12] NetworkX, https://networkx.github.io/documentation/stable/index.html

[13] Python, https://www.python.org/

