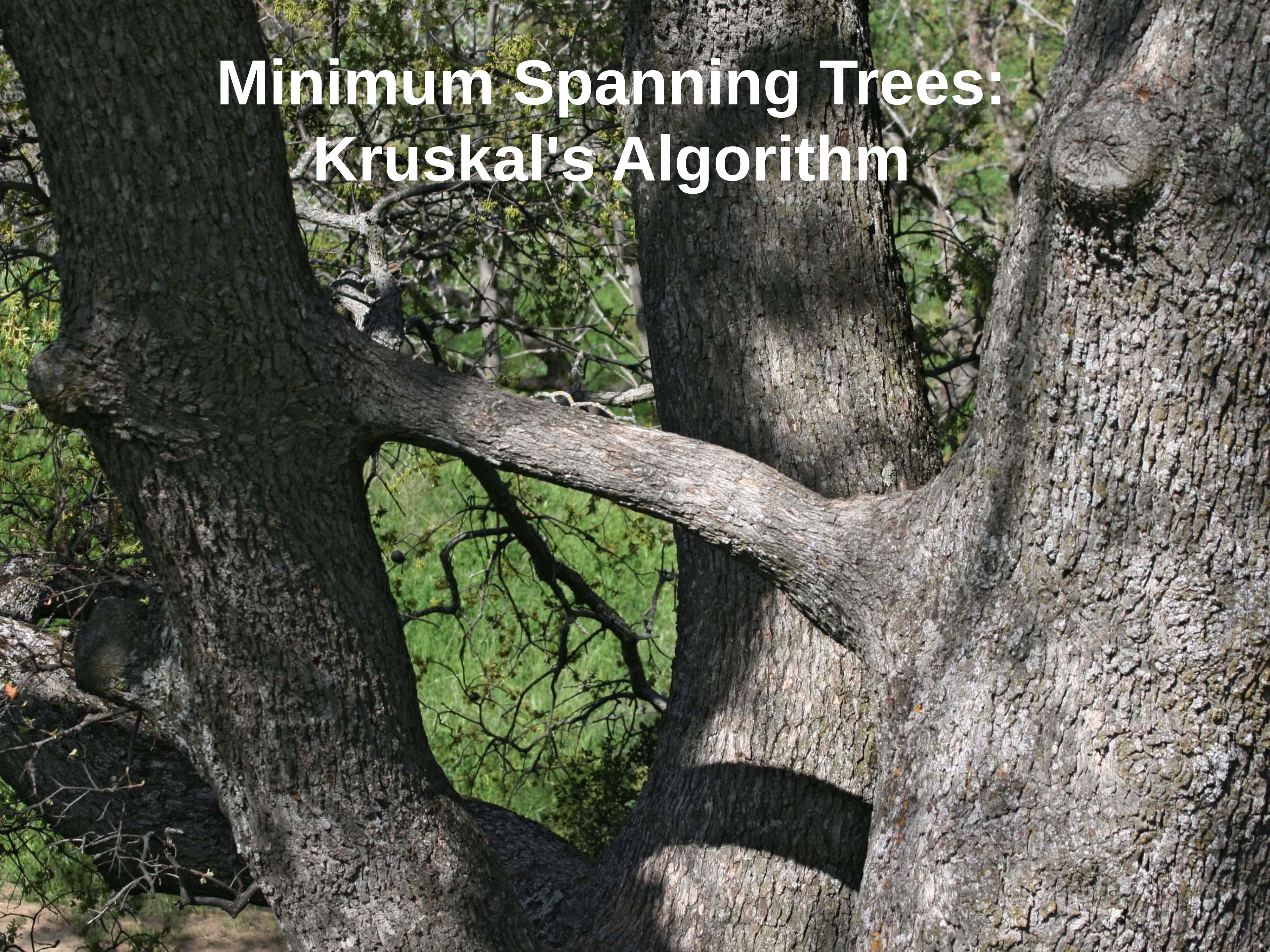
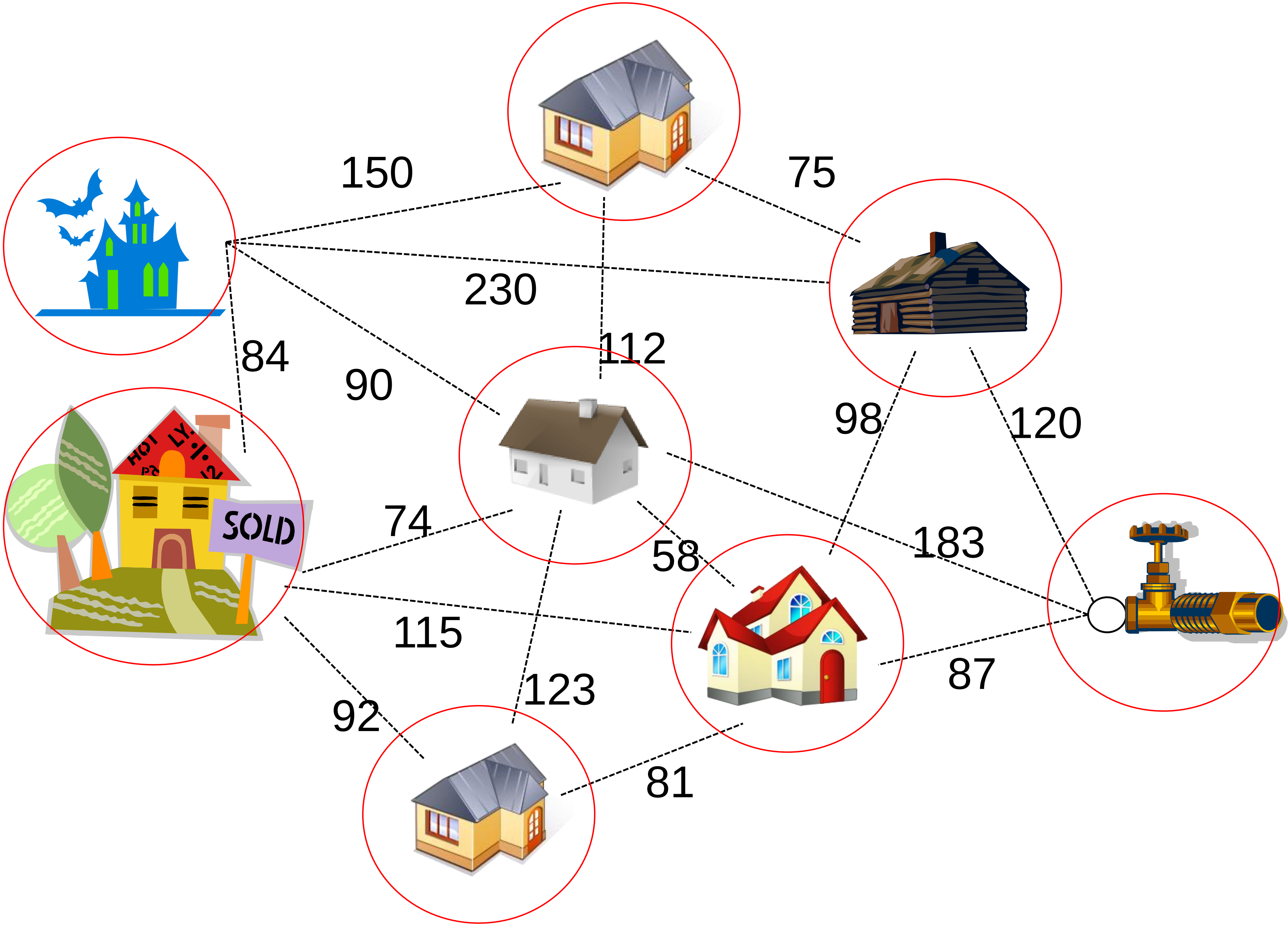
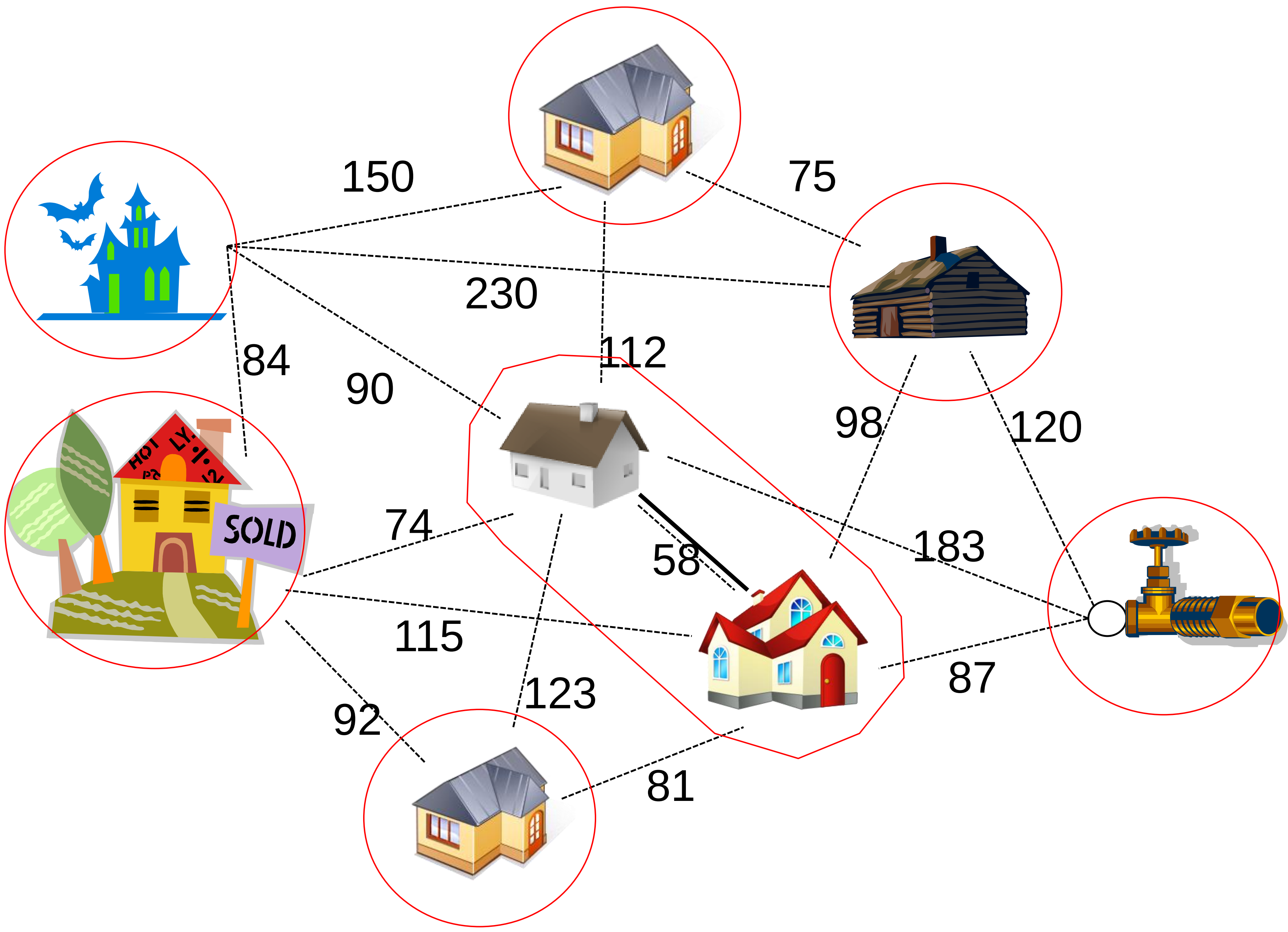

Minimum Spanning Trees:
Kruskal's Algorithm

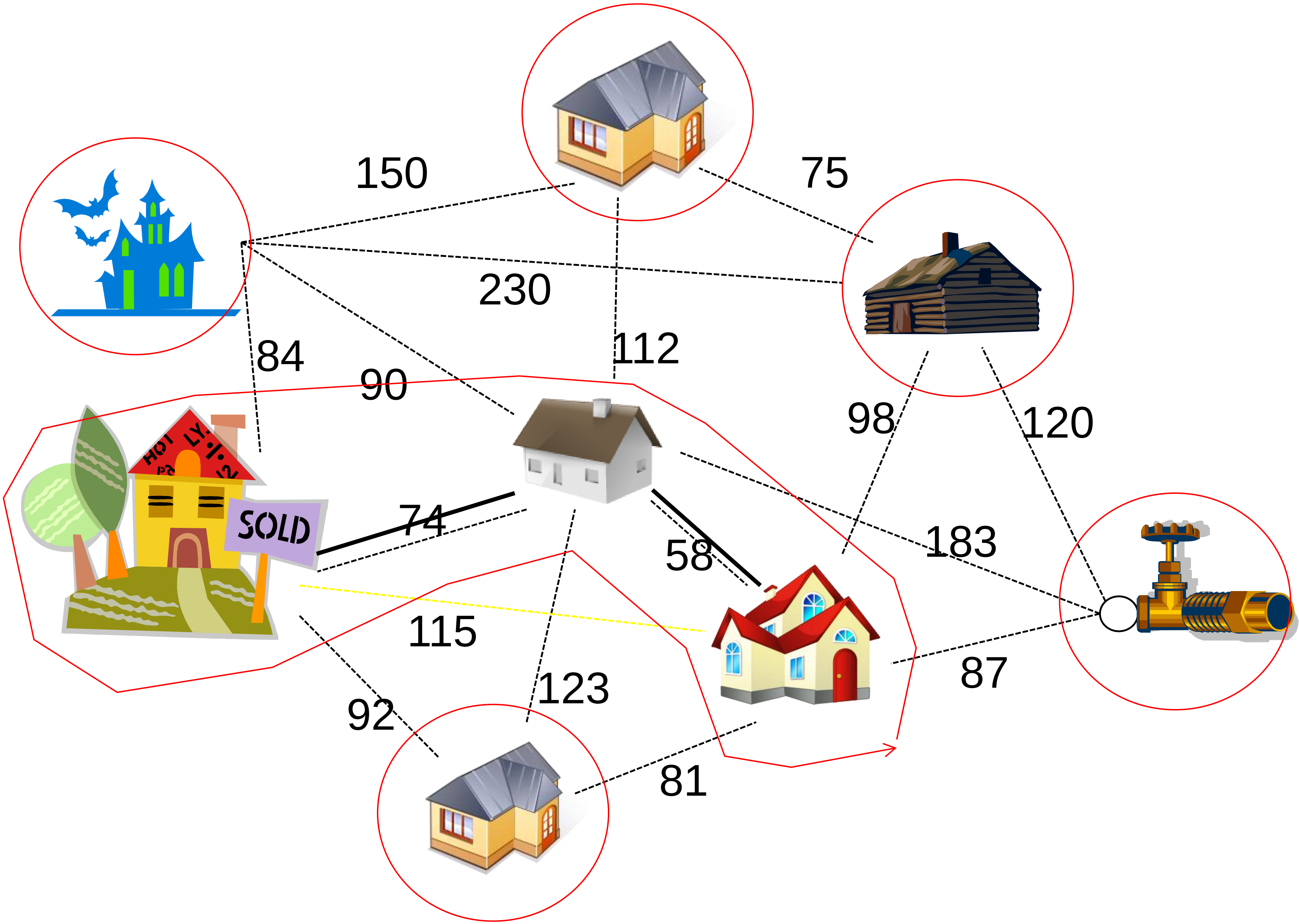
Prim's algorithm worked by gradually building a single tree one edge at a time until all vertices in the graph are in the tree.

There is another approach based on joining trees together until all vertices are in the graph

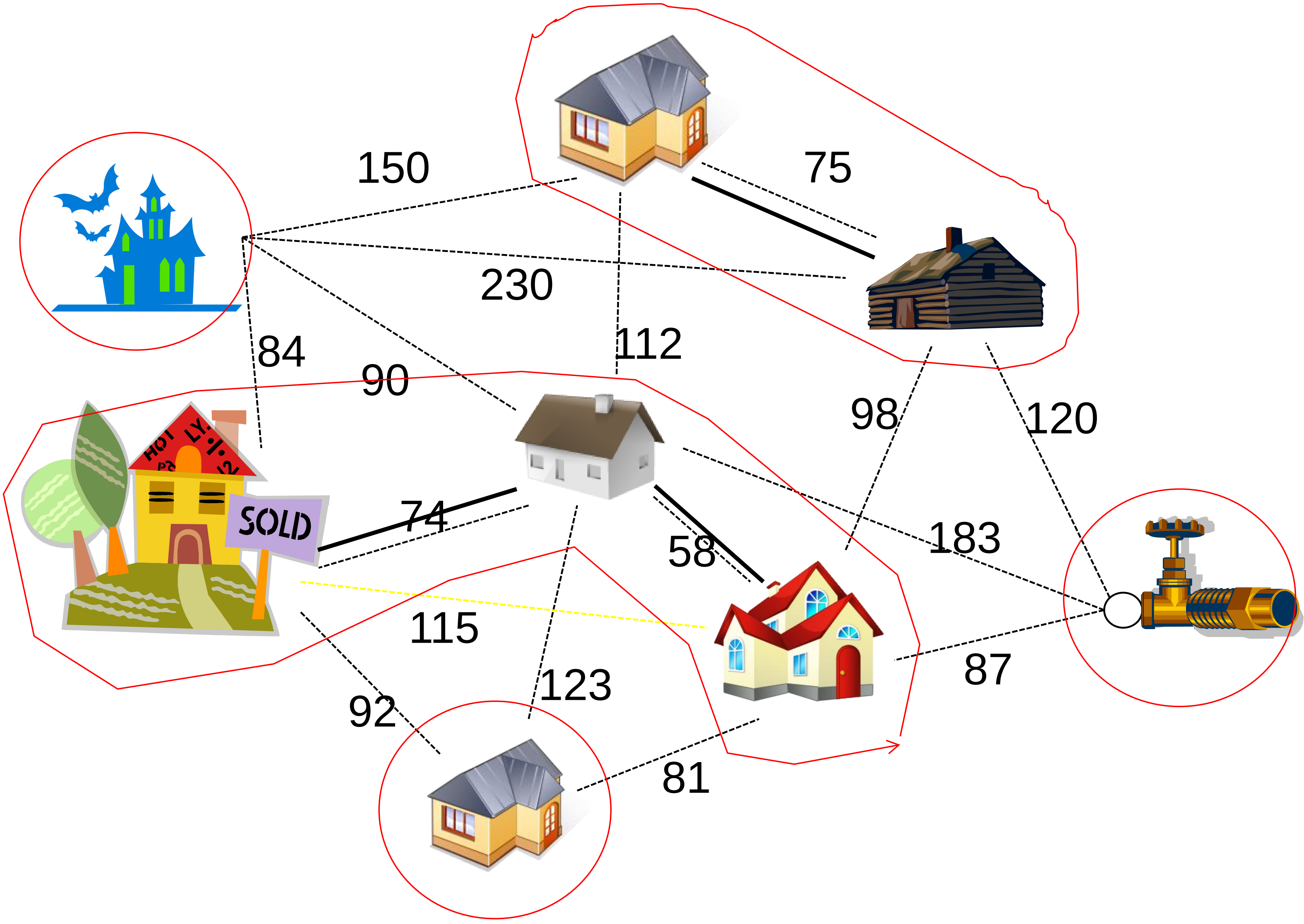

150
75
230
112
84
90
98
120
SOLD
74
58
183
115
123
87
92
81



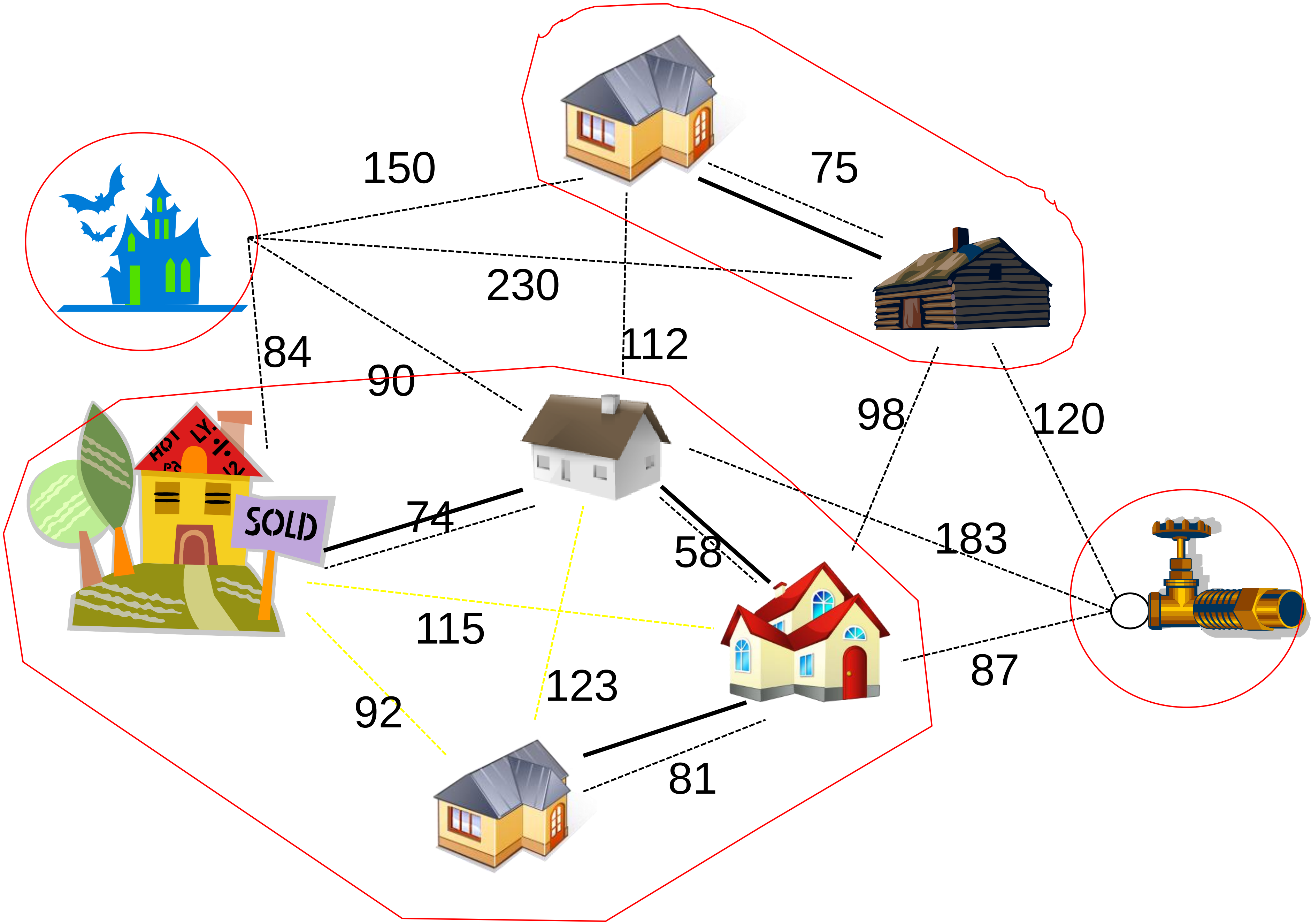
150
75
230
112
84
90
98
120
SOLD
74
58
183
115
87
123
92
81



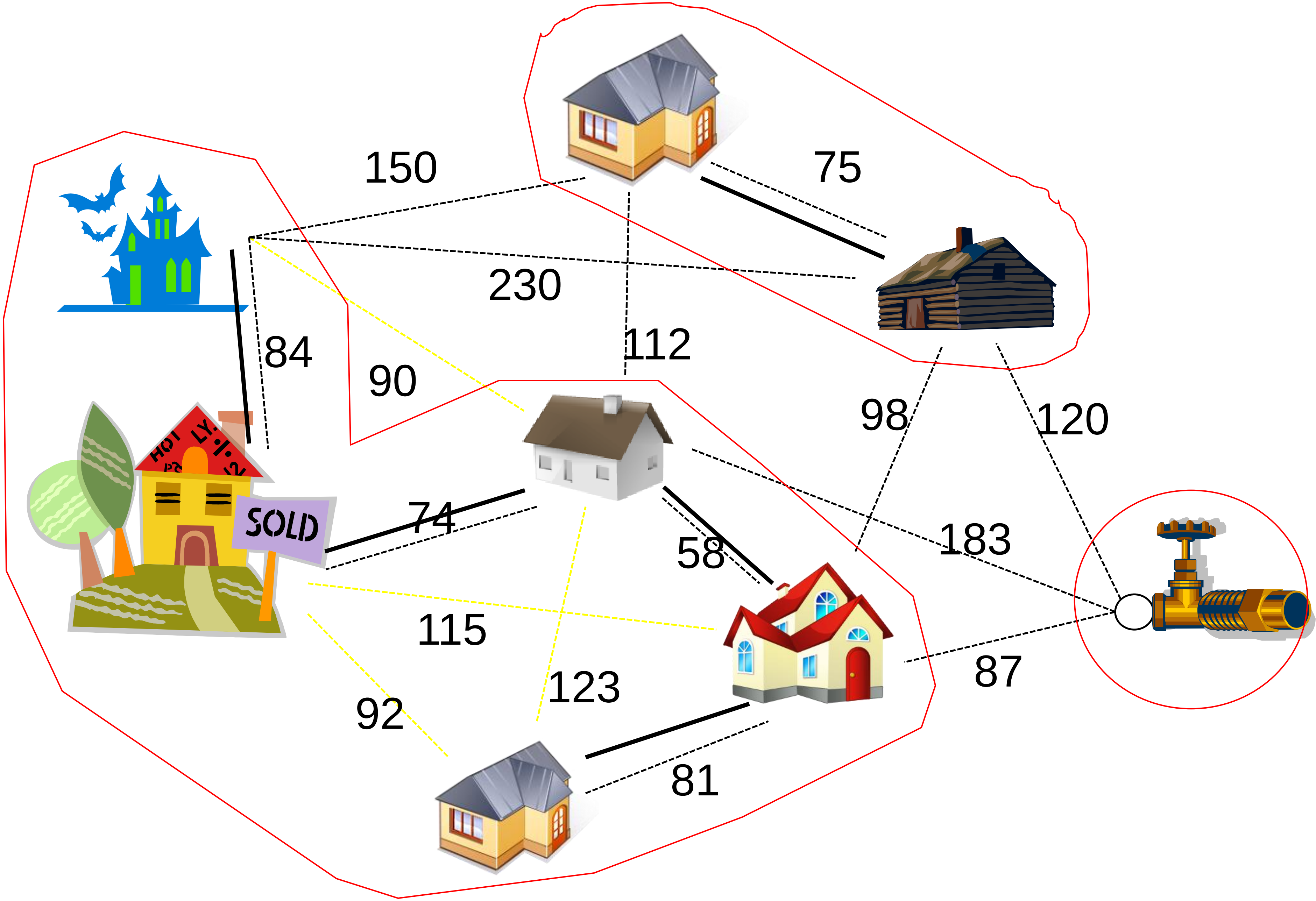
150
75
230
84
90
112
98
120
74
58
183
115
123
92
87
81
SOLD

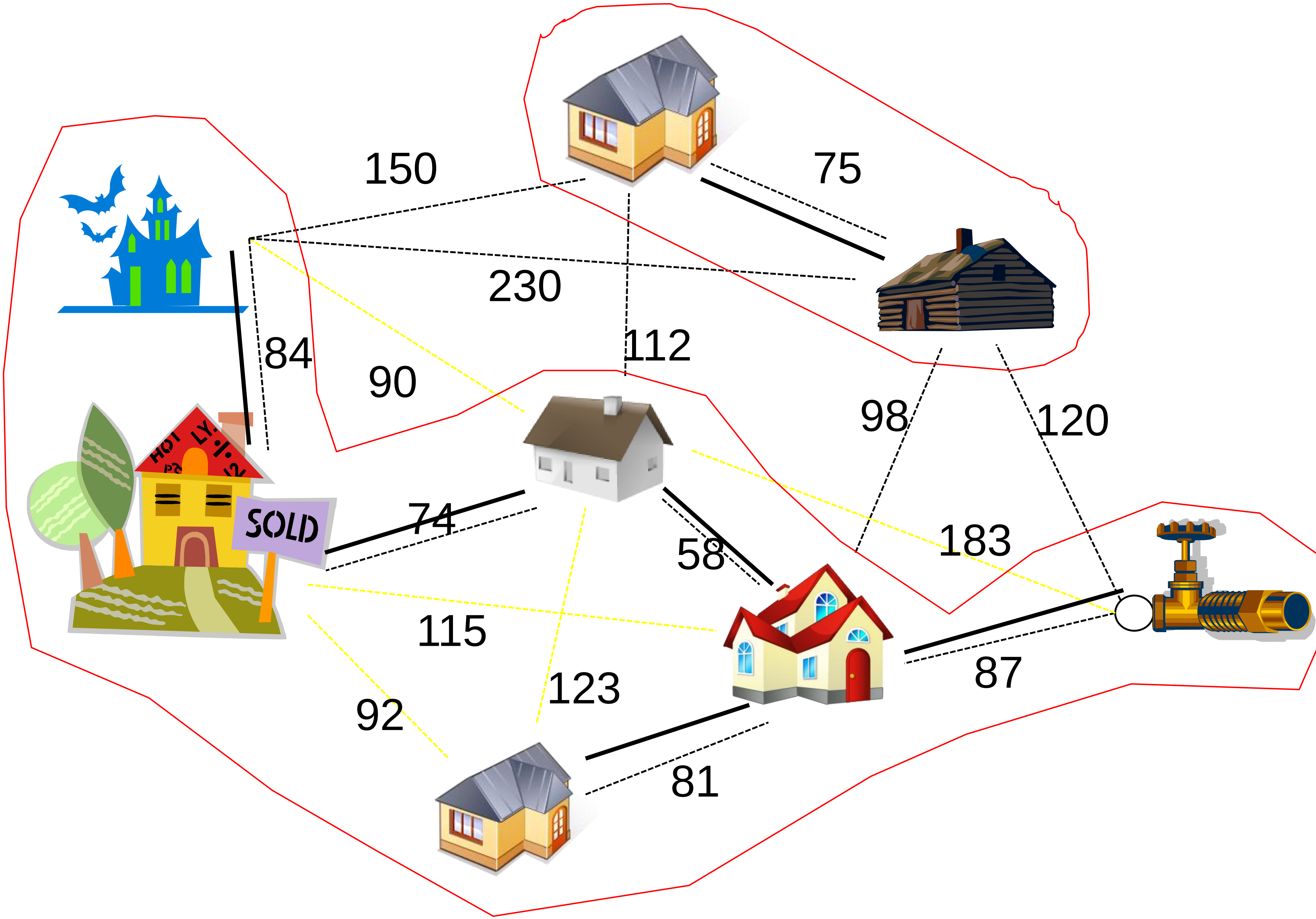


150
75
230
84
90
112
98
120
74
58
183
SOLD
115
123
92
87
81

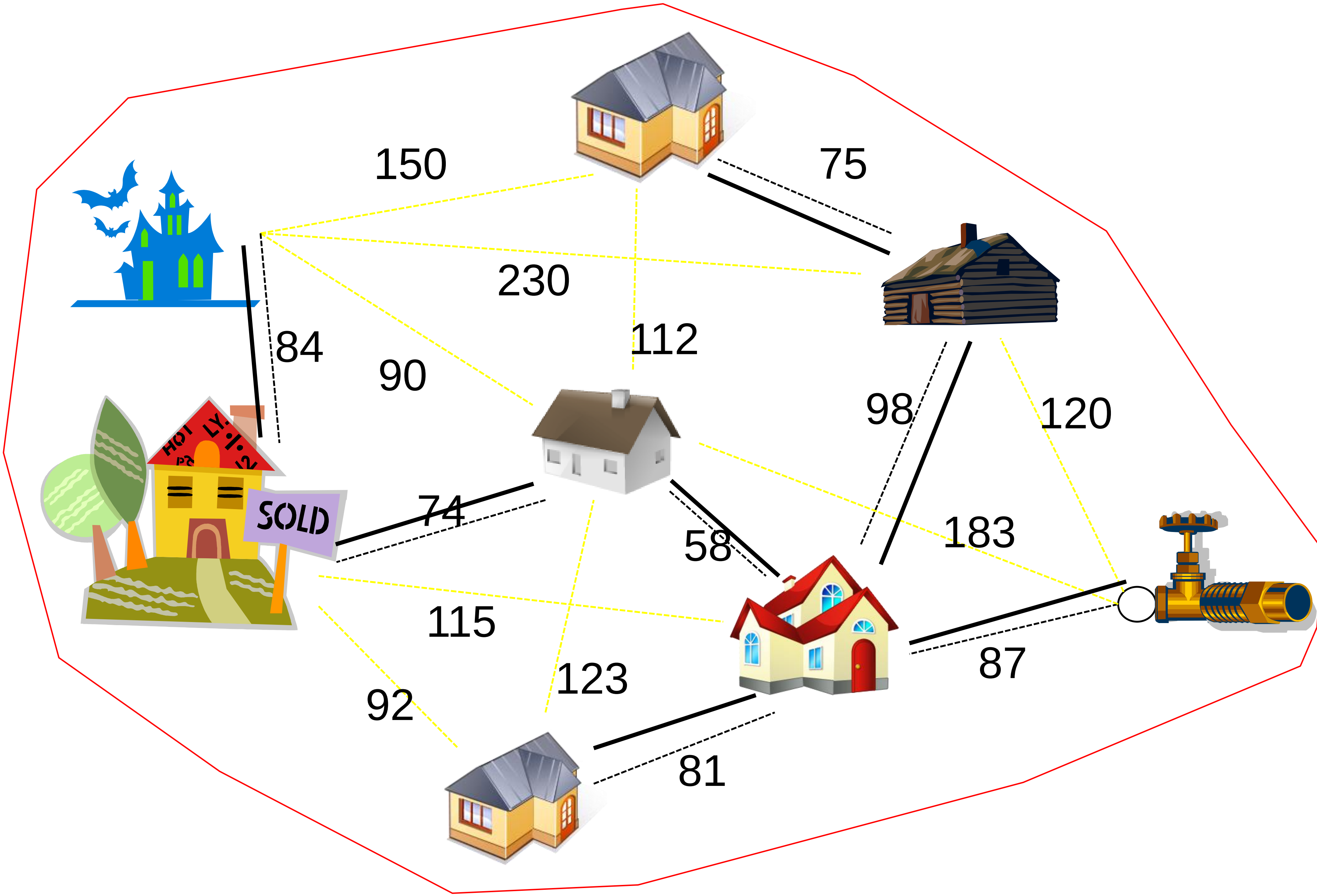


150
75
230
84
90
112
98
120
74
58
183
115
123
92
87
81
SOLD

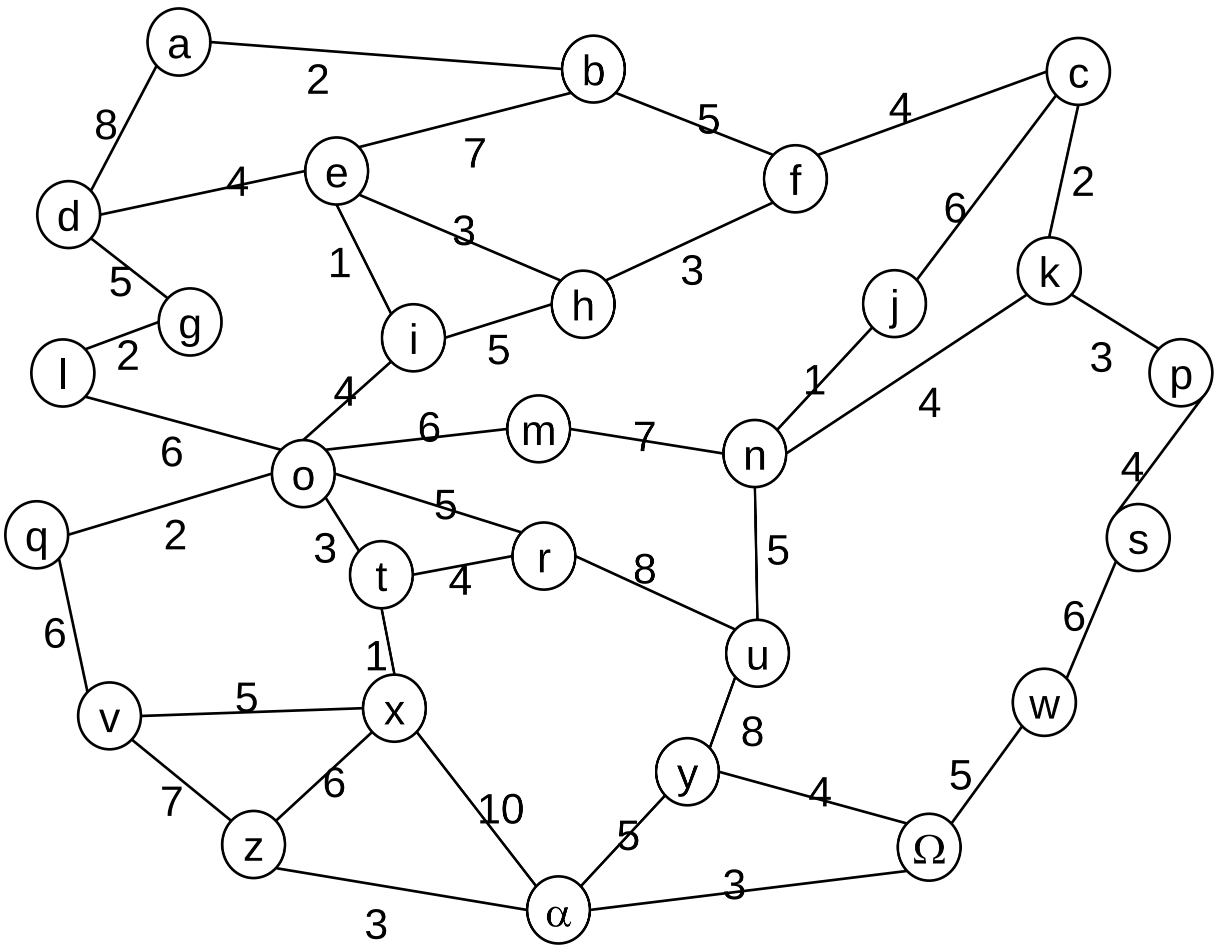

150
75
230
84
90
112
98
120
74
58
183
115
87
123
92
81



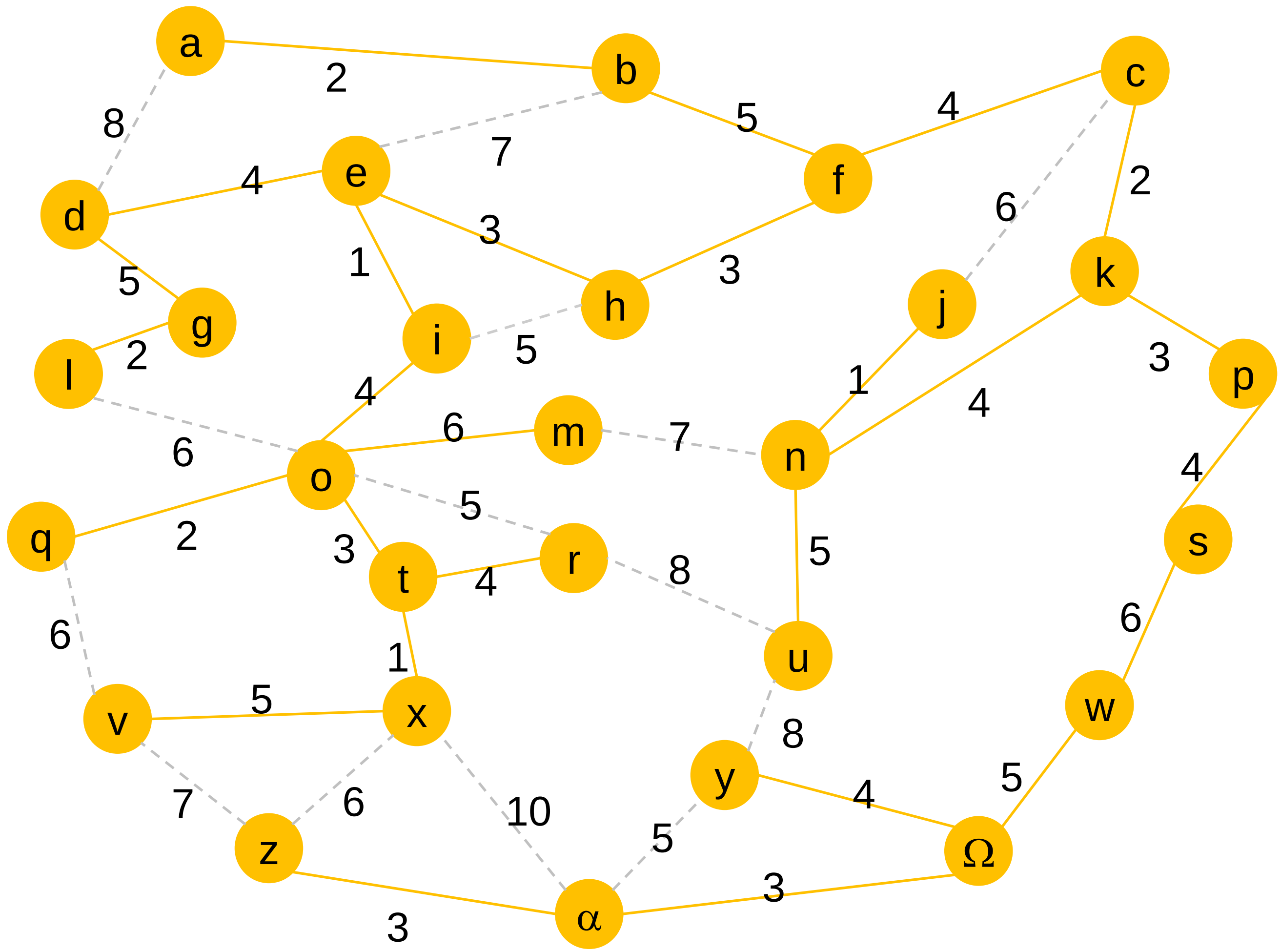
150
75
230
84
90
112
98
120
74
58
183
115
87
123
92
81


150
75
230
84
90
112
98
120
74
58
183
115
87
123
92
81

```
kruskal(): #pseudocode version 1
create an empty mst
for each vertex in the graph
    create a separate tree
for each edge in the graph in increasing order of cost
    if edge joins two separate trees
        add edge to the mst
        merge the two trees into one tree
return mst
```



a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
α
Ω
2
5
4
8
7
4
6
2
3
1
3
5
5
2
1
4
5
3
4
6
6
7
4
4
2
5
3
3
8
5
4
6
1
5
6
8
5
6
7
10
5
4
3
3



a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
α
Ω
2
8
7
5
4
4
1
3
6
2
5
2
3
5
4
3
1
6
4
6
7
4
4
5
2
3
5
8
5
6
6
5
1
6
7
6
10
8
4
5
5
3
3

```
kruskal(): #pseudocode version 1
create an empty mst
for each vertex in the graph
    create a separate tree
for each edge in the graph in increasing order of cost
    if edge joins two separate trees
        add edge to the mst
        merge the two trees into one tree
return mst
```

How do we implement this efficiently?

```
kruskal(): #pseudocode version 2
create an empty mst
create an empty dictionary stating the tree containing each vertex
create an APQ pq
for each edge e in G
    add (w(e), e) into pq
for each vertex v in G
    create a stack s with v as only element
    store s as value for v in dictionary
while length(mst) < |V| and pq is not empty
    remove the minimum cost edge
    get the trees of its vertices from the dictionary
    if the two trees are different
        add the edge into mst
        join the two trees into a single tree and update dictionary
return mst
```

How do we implement this efficiently?

Representing the trees

Implement each tree as a stack (or any sequence).

To join two trees, find the smaller one, then pop each element, update its dictionary to the bigger tree, and push onto the bigger tree.

Note: more efficient implementations of tree merging can give better complexity

Complexity:

creating the dictionary: $O(n)$
creating the PQ: $O(m \log m)$ - the number of edges
at most m times round the loop
for each time, $O(\log m)$ to remove the min-cost edge, $O(1)$ to get the trees for the two vertices and test if they are different, and then the cost of updating the dictionary and merge the trees.

So $O(m \log m)$ overall to handle the PQ, plus the cost of the tree merges.
The tree merging only happens n-1 times.

Each time two trees get merged, we create a tree at least twice the size of the smaller tree. So each vertex can change trees at most $O(\log n)$ times.
So amortised cost of tree merging is $O(n \log n)$.

So in total $O(n \log n + m \log m)$
But m is $O(n^2)$, so $O(\log m) = O(\log n^2) = O(2 \log n) = O(\log n)$
So algorithm is $O((n+m)\log n)$

```python
def mst_k(self):
    tree = []
    n = self.num_vertices()
    whichtree = {}

    for v in self.vertices():
        vtree = Stack()
        vtree.push(v)
        whichtree[v] = vtree
    pq = PQHeap()

    for e in self.edges():
        pq.add(e.element(), e)
    while len(tree) < n and not pq.is_empty():
        key, e = pq.remove_min()
        (x,y) = e.vertices()
        xtree = whichtree[x]
        ytree = whichtree[y]
        if xtree != ytree:
            tree.append(e)
            self._jointrees(xtree, ytree, whichtree)
    return tree
```

```python
def _jointrees(self, xtree, ytree, whichtree):
    if xtree.length() < ytree.length():
        target = ytree
        deltree = xtree
    else:
        target = xtree
        deltree = ytree
    while deltree.length() > 0:
        v = deltree.pop()
        whichtree[v] = target
        target.push(v)
    del deltree
```

Next lecture

Further graph algorithms