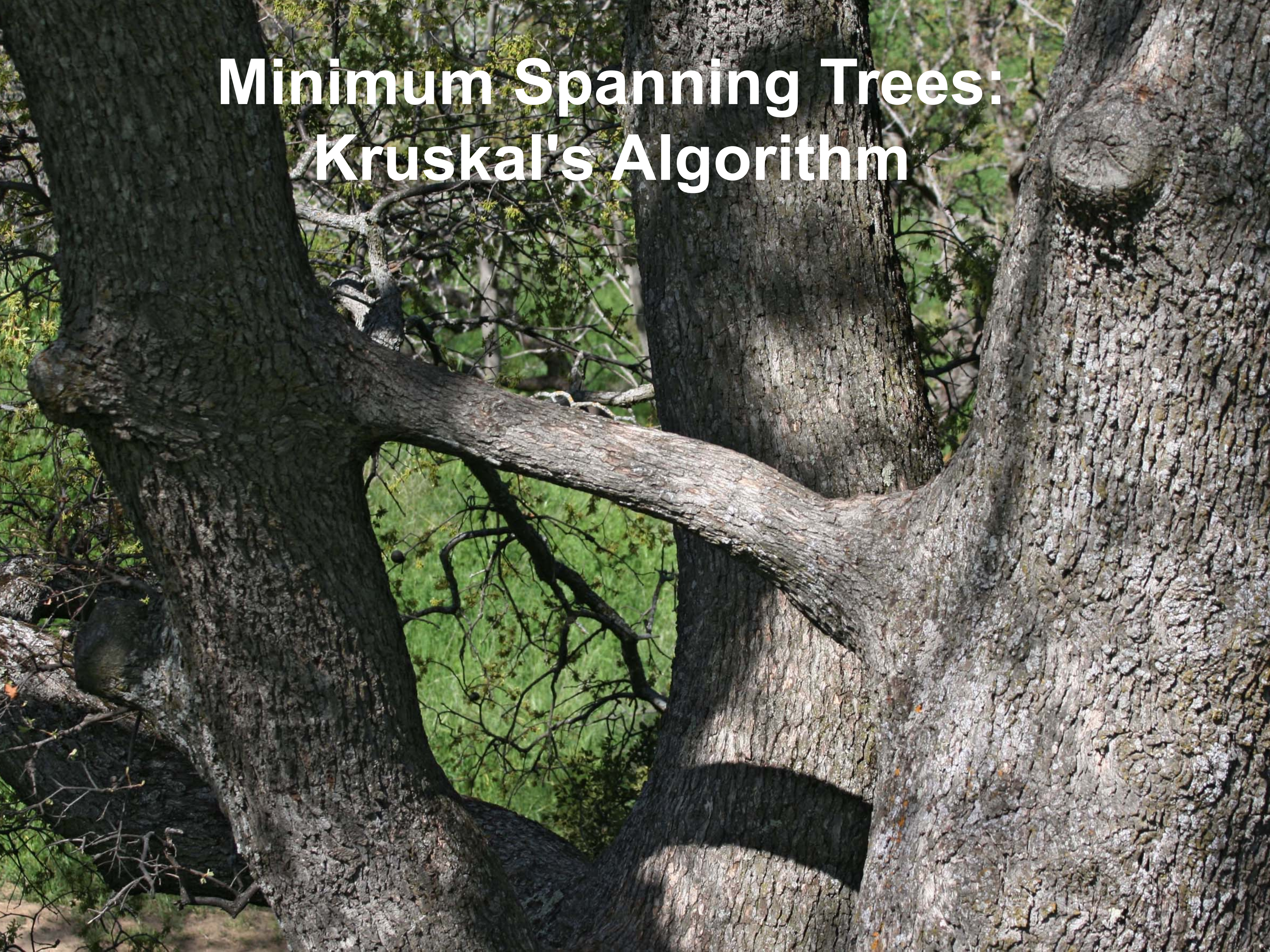
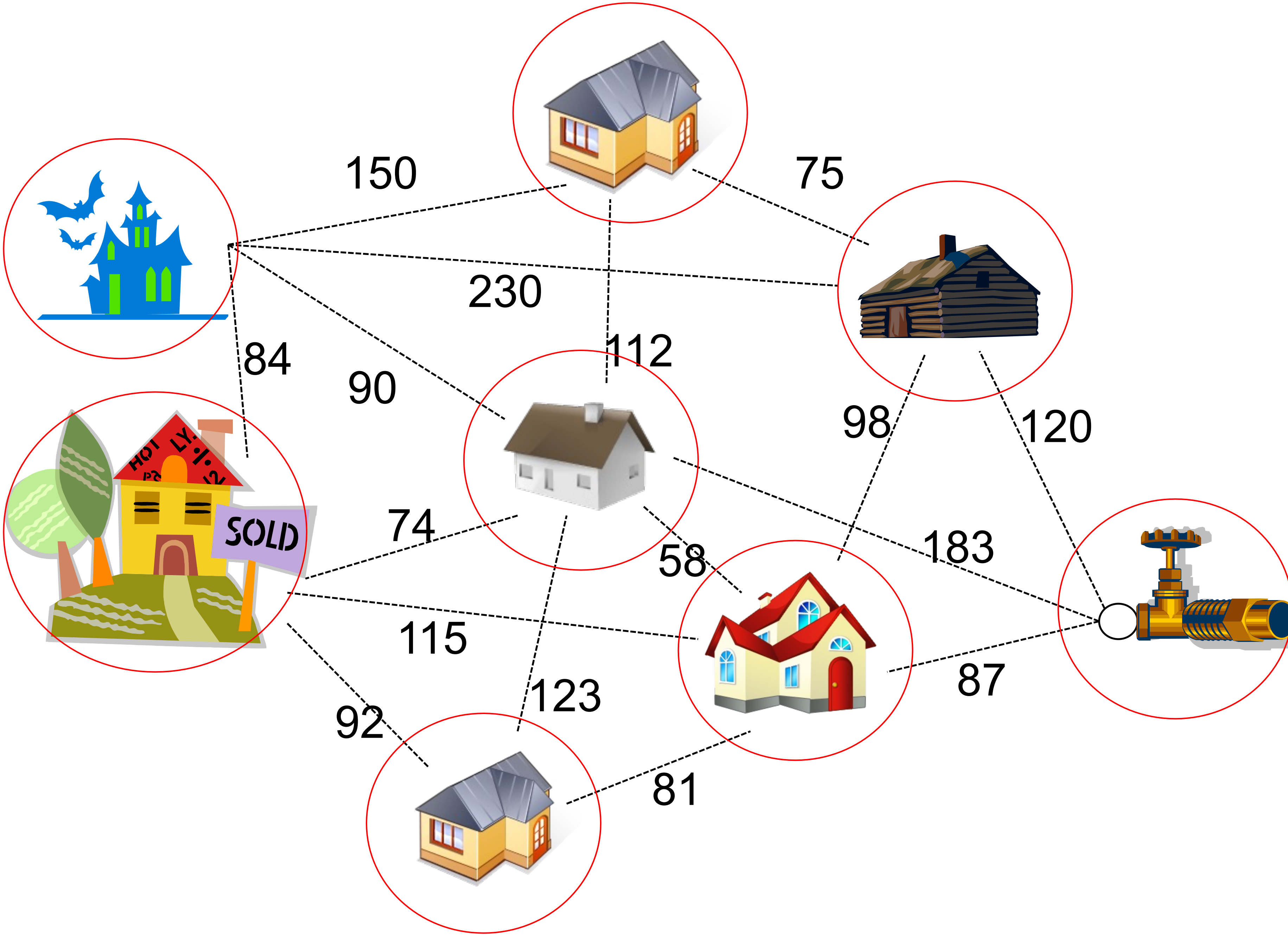
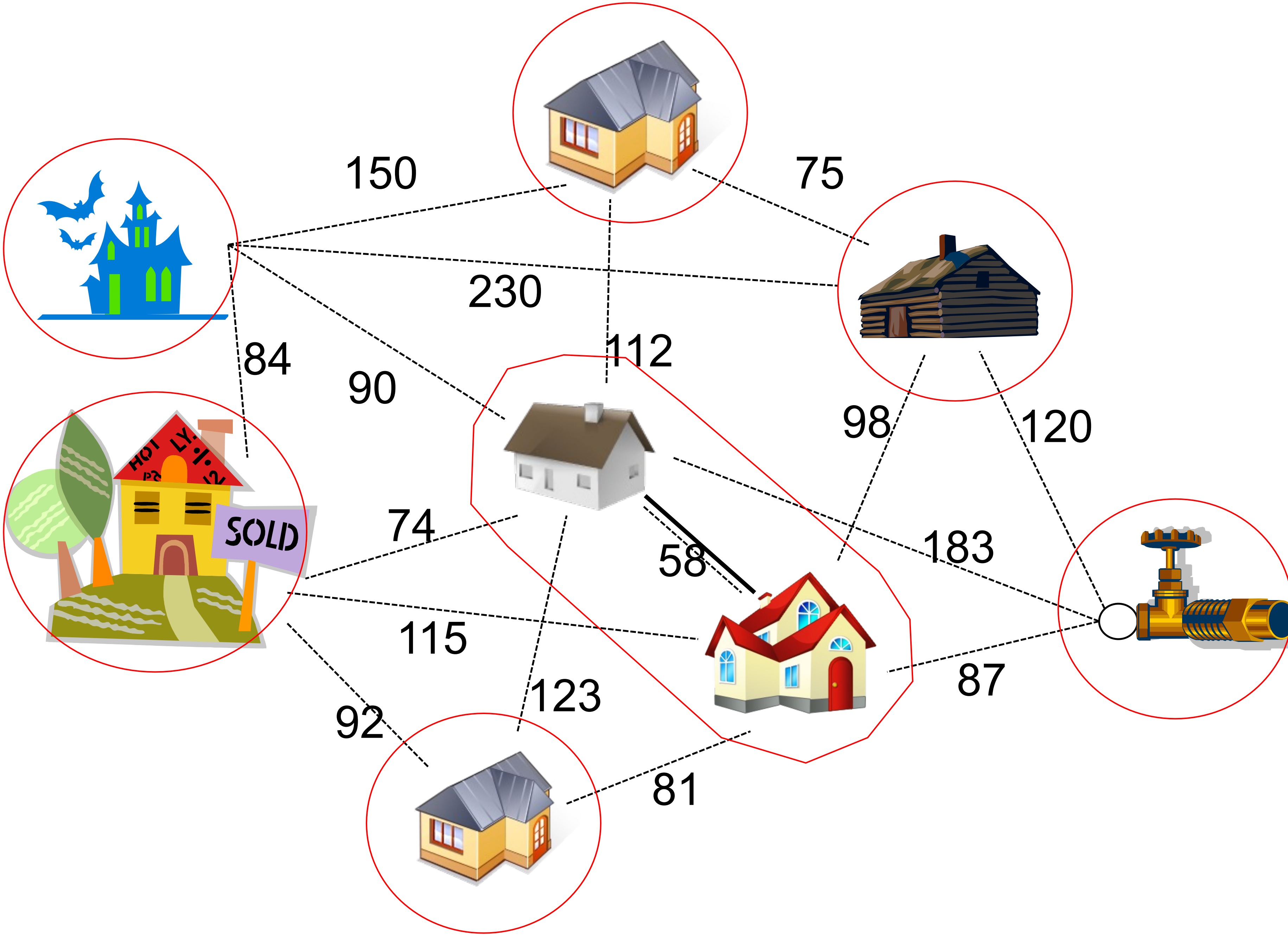
# Minimum Spanning Trees: Kruskal's Algorithm

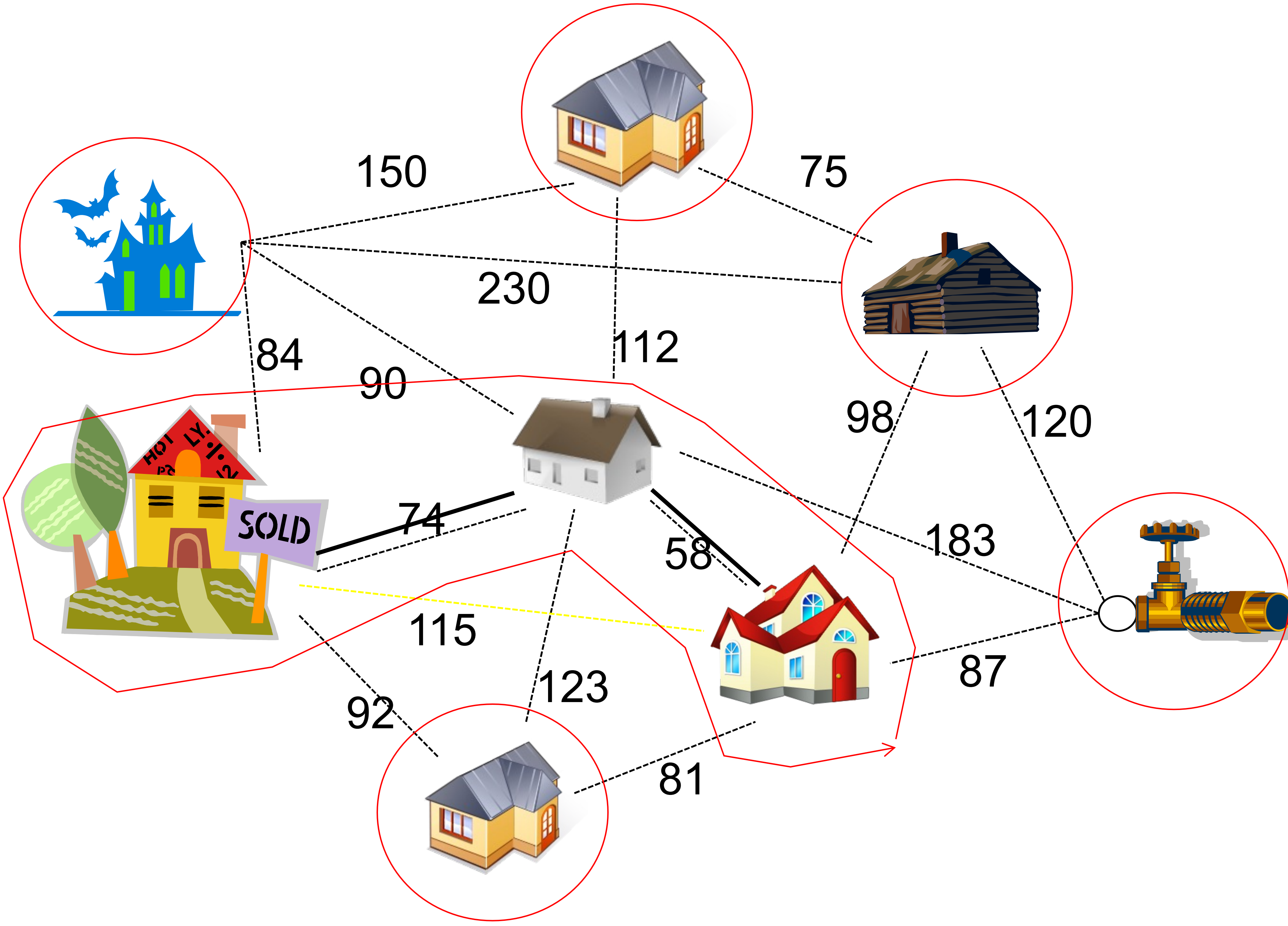


Prim's algorithm worked by gradually building a single tree one edge at a time until all vertices in the graph are in the tree.

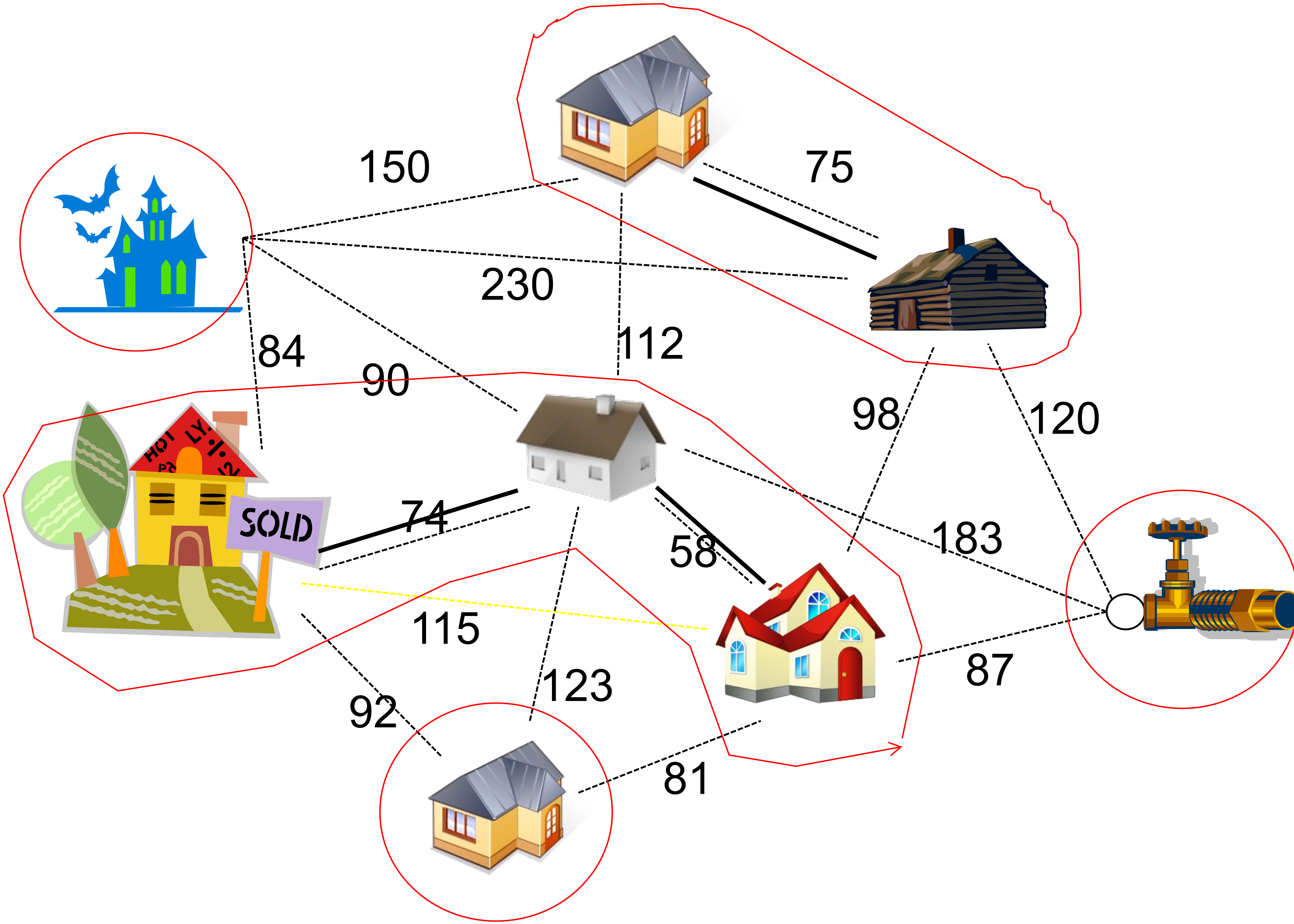
There is another approach, Kruskal's algorithm, which works by joining trees together until all vertices from the graph are in a single tree.

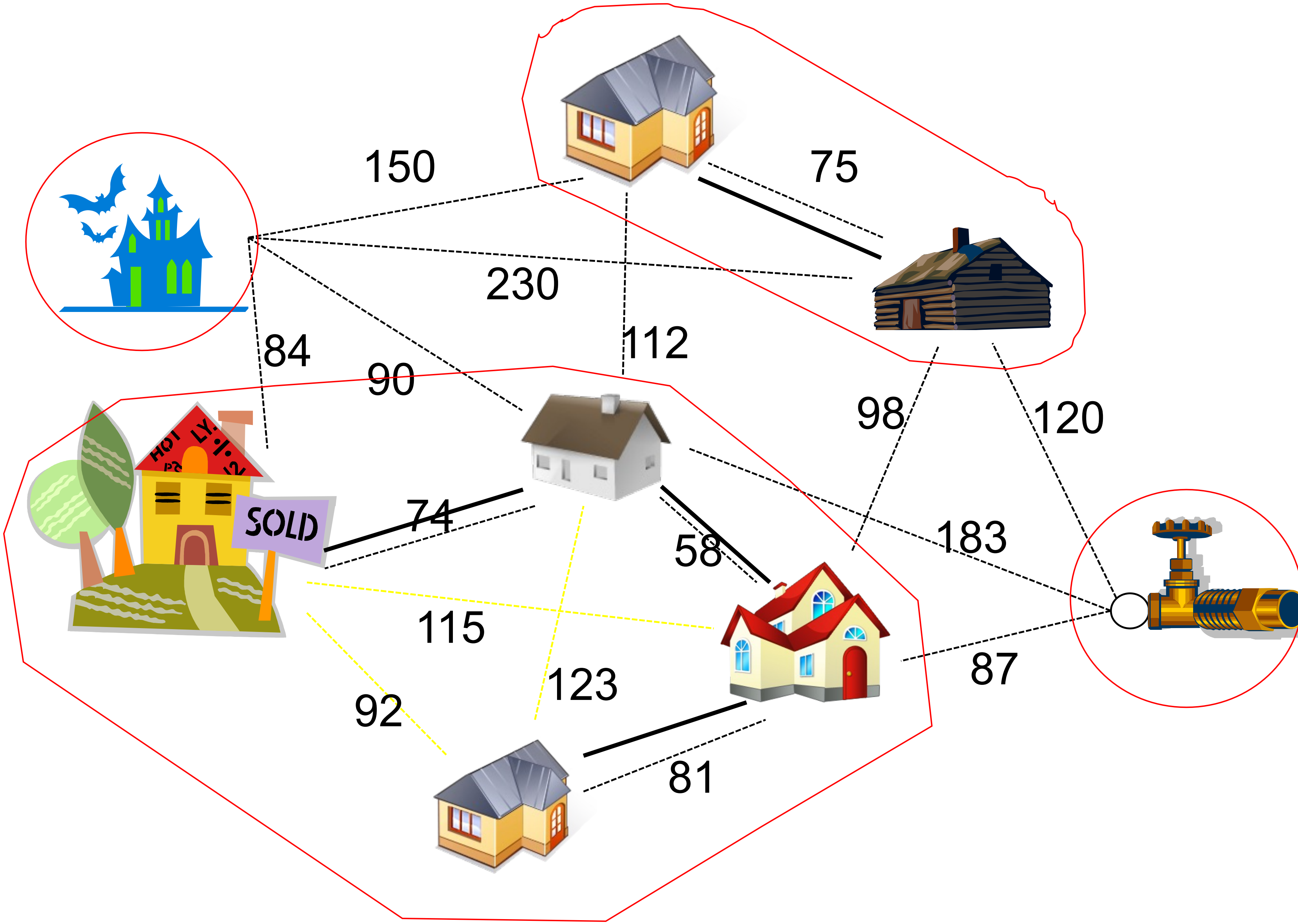

150
75
230
112
84
90
98
120
74
58
183
115
87
123
92
81
SOLD

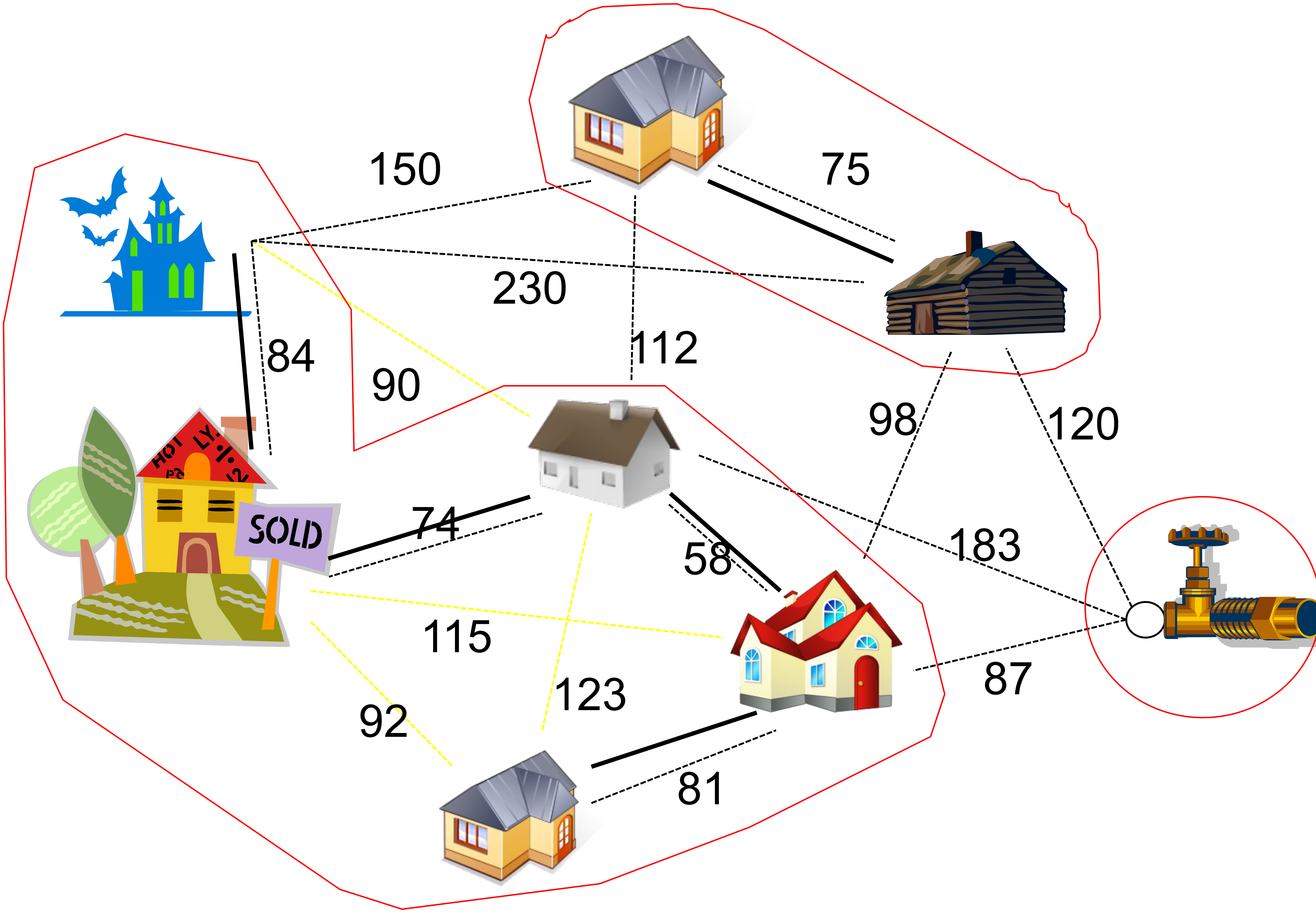

150
75
230
112
84
90
98
120
SOLD
74
58
183
115
87
123
92
81

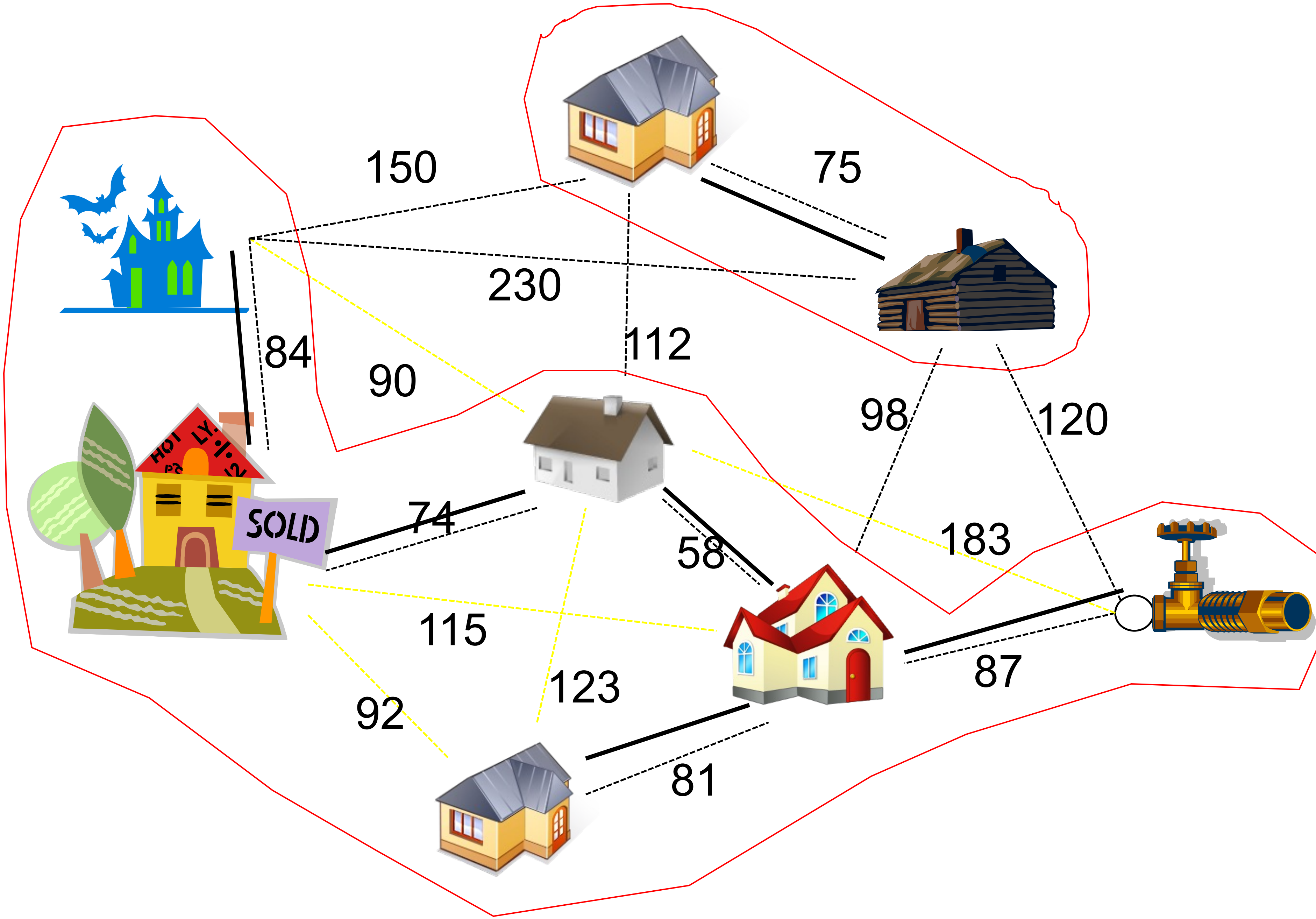


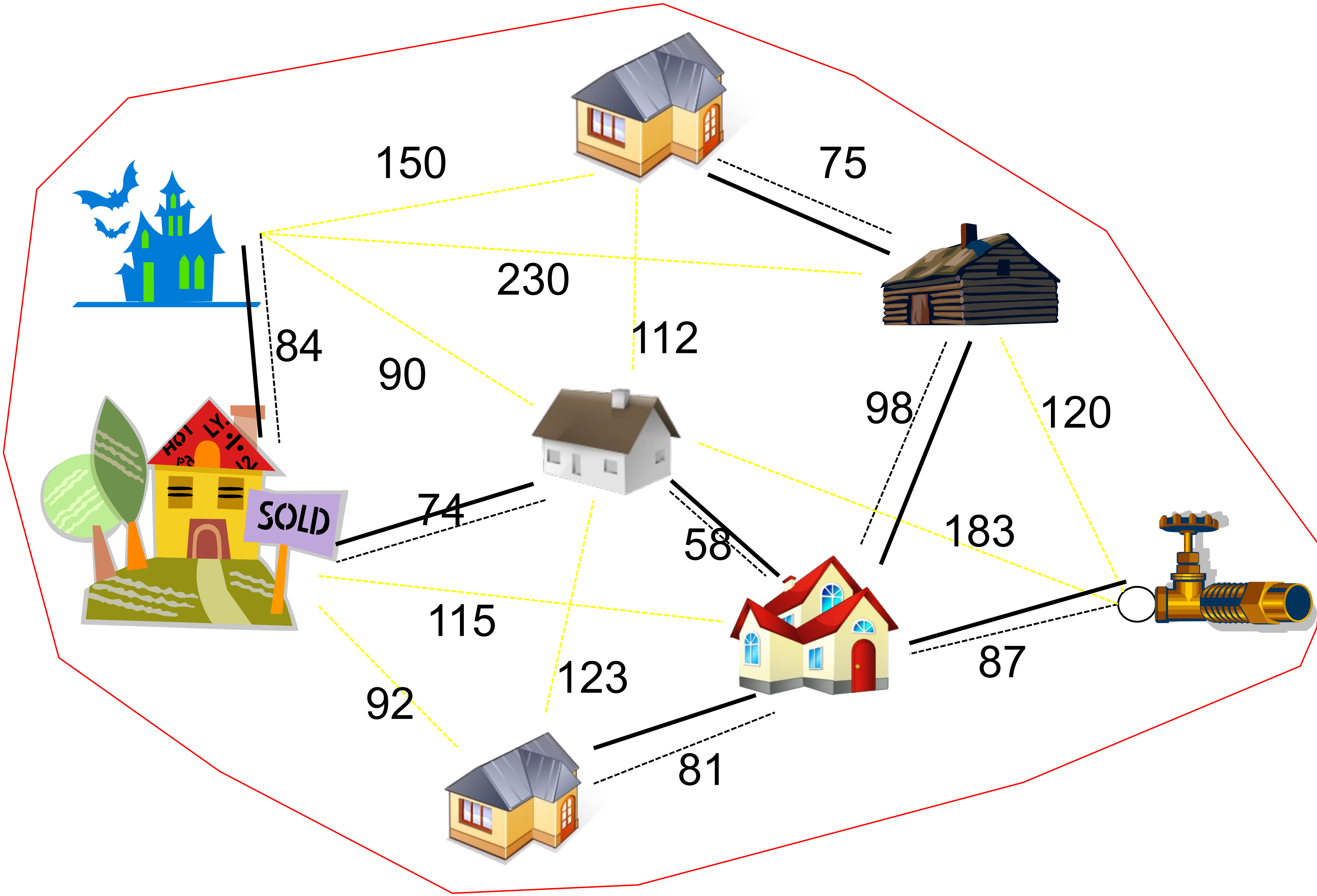
150
75
230
112
84
90
98
120
74
58
183
SOLD
115
87
123
92
81


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


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


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

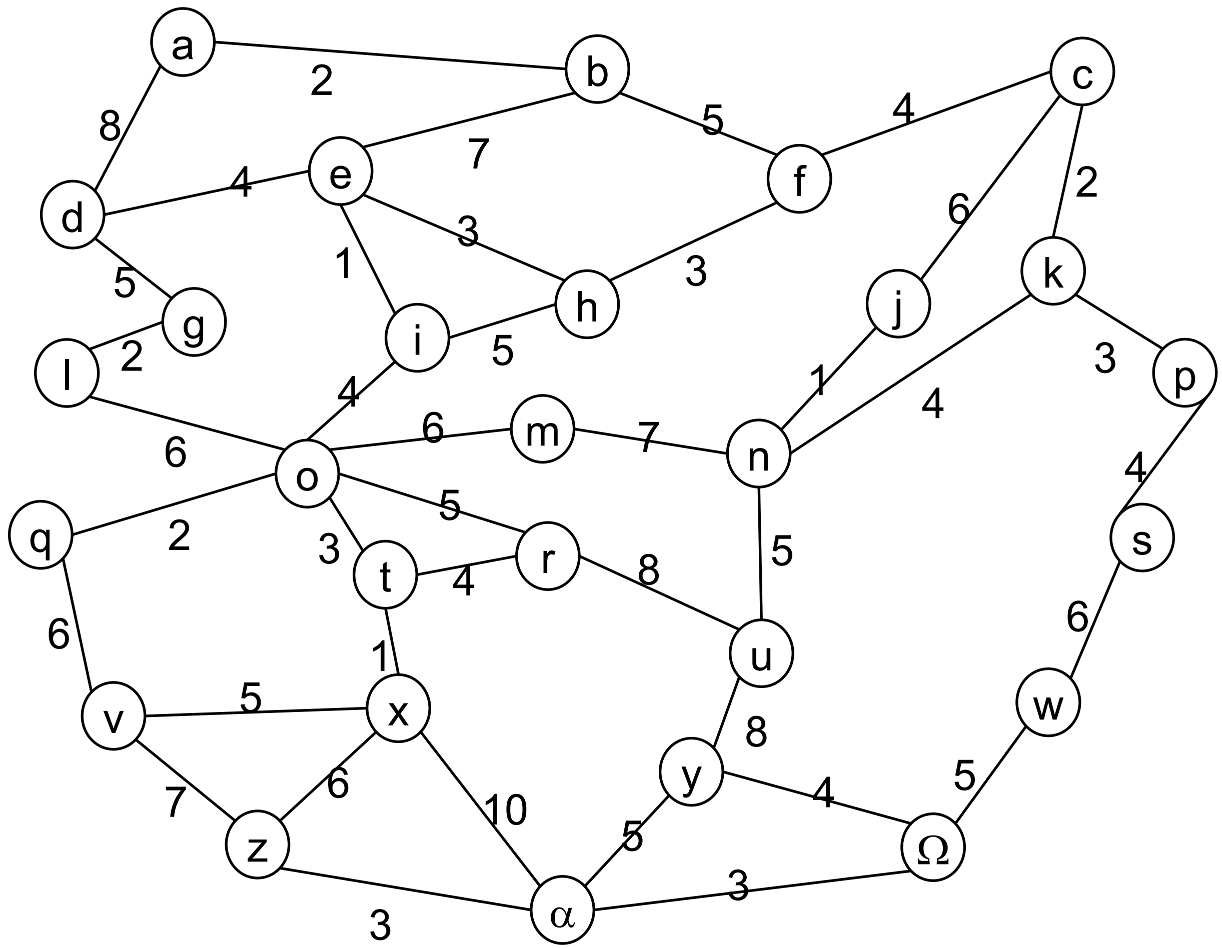

150
75
230
84
90
112
98
120
SOLD
74
58
183
115
123
87
92
81

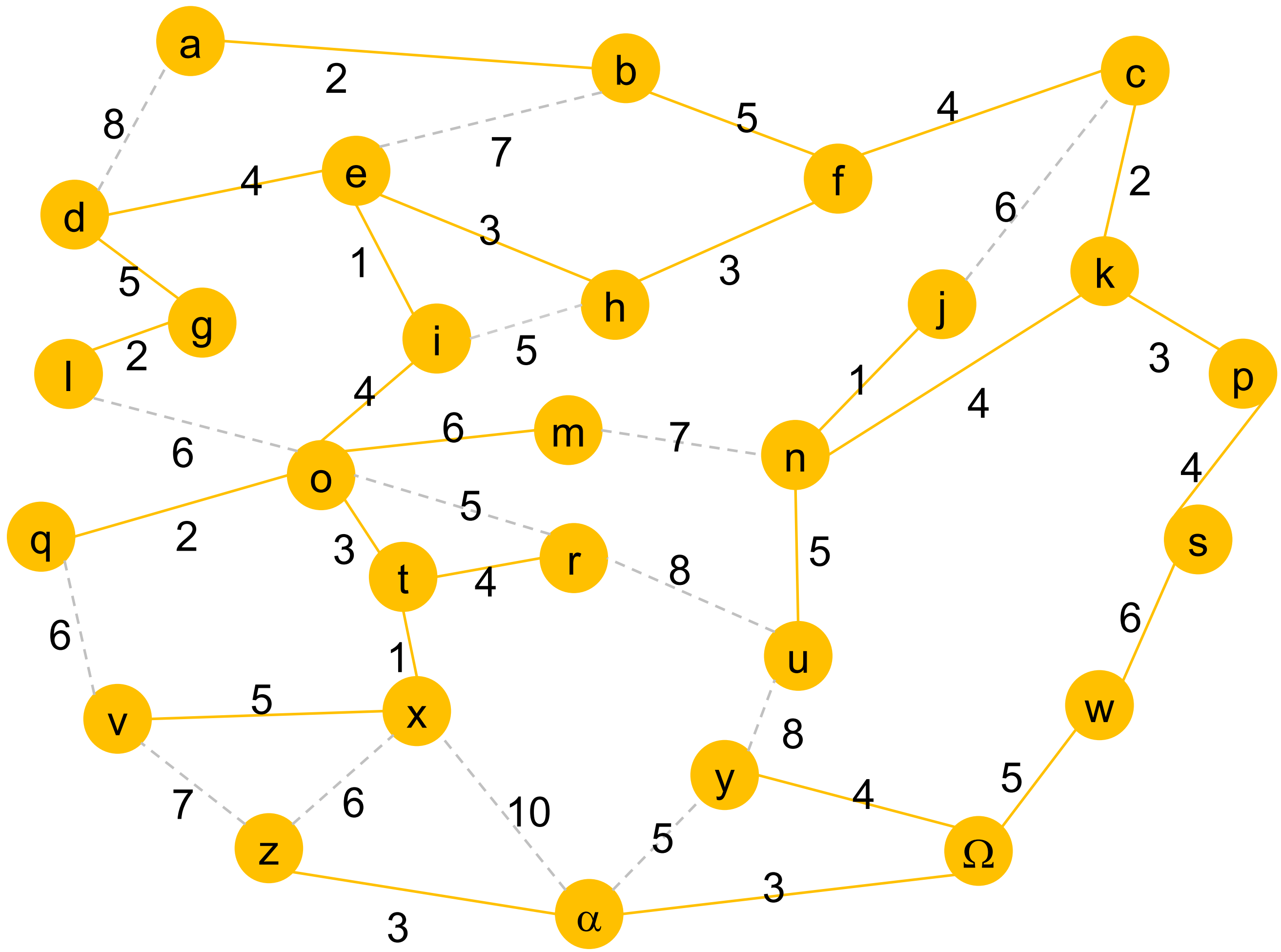

150
75
230
112
84
90
98
120
74
58
183
115
87
123
92
81
SOLD

```
kruskal(): #pseudocode version 1
create an empty mst  //a list of edges
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
```

Sketch of proof of correctness

- if the graph is connected, ‘mst’ contains every vertex, so spans the graph
- ‘mst’ is a tree (never adds an edge that creates a cycle)
- proof that it has minimum cost is similar to that for Prim’s algorithm


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
4
7
5
4
3
1
3
6
2
5
2
5
4
6
6
7
1
4
3
4
2
6
5
3
4
8
5
6
5
1
6
7
6
10
8
5
4
5
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
2
1
3
6
5
3
2
5
1
3
4
4
6
6
7
4
2
5
5
4
3
8
6
5
1
6
8
5
7
6
10
5
4
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 list mst        // to contain edges in final MST
create an empty dictionary  // for each vertex, state its current tree
create a PriorityQueue pq   //to order graph edges by cost
for each edge e in G
   add (w(e), e) into pq
for each vertex v in G
   create a tree t for v and add (v:t) to dictionary
while length(mst) < |V|-1 and pq is not empty
   remove e,  the minimum cost edge from pq
   get the trees t1, t2 for e's vertices from dictionary
   if t1 and t2 are different
      add e into mst
      join t1 and t2 into a single tree t and update dictionary
return mst
```

How do we implement this efficiently?

# Representing the trees

Implement each tree as a stack (or any sequence with O(1) updates).

To join two trees, find the smaller one, then until that stack is empty:

- pop a vertex
- update its dictionary entry to point to the bigger tree
- push onto the bigger tree.

```python
def kruskal(self):
    mst = []
    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(mst) < n and pq.length() > 0:
        key, e = pq.remove_min()
        (x,y) = e.vertices()
        xtree = whichtree[x]
        ytree = whichtree[y]
        if xtree != ytree:
            mst.append(e)
            self._jointrees(xtree, ytree, whichtree)
    return mst
```

```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
```

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 around the loop, $O(\log m)$ 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 merging the trees.

So $O(m \log m)$ overall to handle the PQ, ignoring the tree merge cost.

Each time two trees get merged, the vertices in the smaller tree move. Each individual vertex move is $O(1)$ – pop(), dictionary update, push(). Since we only move vertices from the smaller tree, each vertex that is moved moves to a tree that becomes at least double the size. There are *n* vertices, so each vertex can change trees at most $O(\log n)$ times (since 1*2*2*...*2 for log n times is n). 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 $O((n+m)\log n)$

doesn't matter if it is sparse or dense ...

# Next lecture

Further graph algorithms