159.271
Computational Thinking for Problem Solving

Greedy Algorithms

Catherine McCartin
c.m.mccartin@massey.ac.nz

Greedy Algorithms

build up a solution to a problem by making a single simple choice at every step

contrast with recursive backtracking:

try out all possible choices recursively

for some problems works well

either produces the optimal solution,

or a reasonably good solution, but not optimal

Greedy coin changing

make change for amount A, using fewest coins possible, given a set of available coin sizes

greedy rule: select largest coin that will fit

example:

given 10, 5 and 1c coins, make change for 18c

choose 10, then 5, then three 1c coins

gives an optimal solution

Greedy coin changing

example:

given 10, 6 and 1c coins, make change for 12c

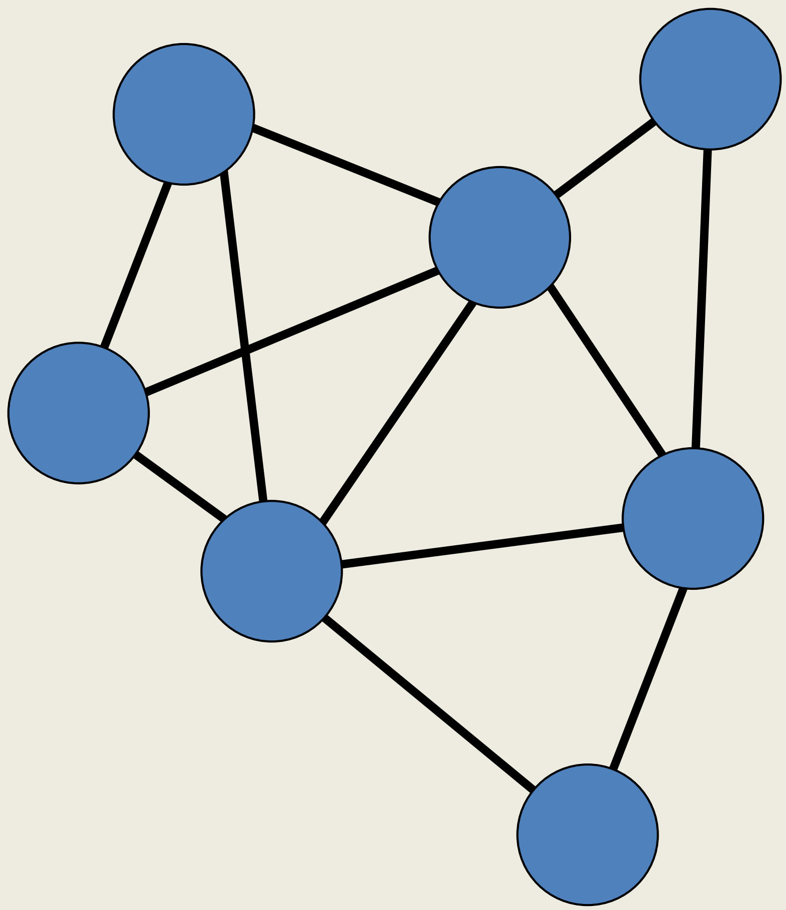
choose 10, then two 1c coins

doesn't give an optimal solution

when does this greedy rule give an optimal solution?

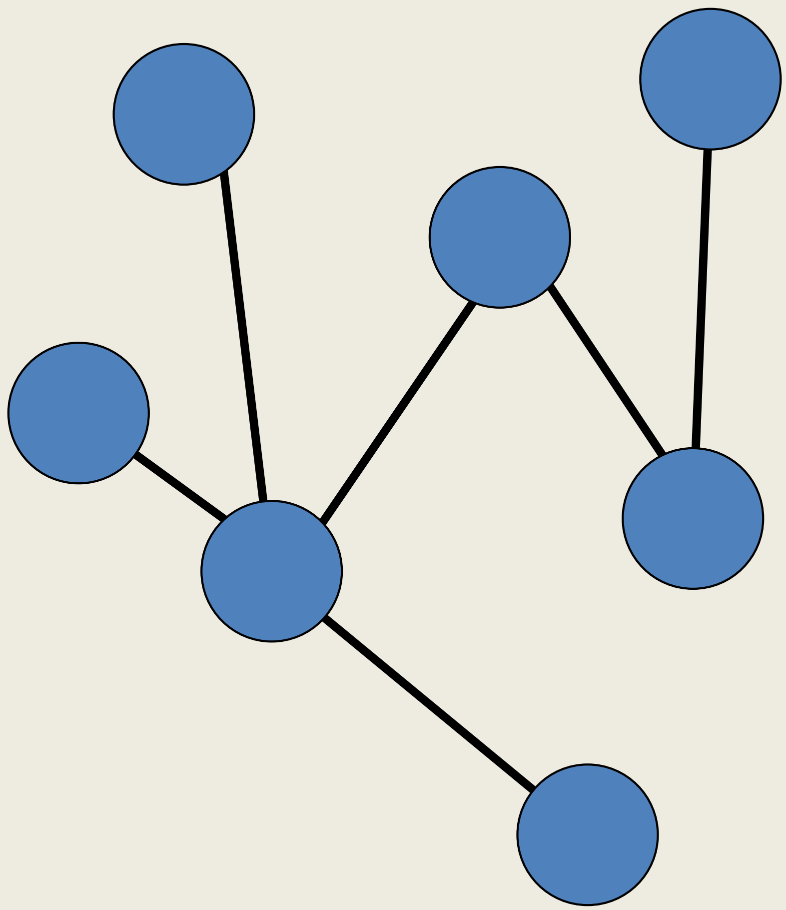
Spanning trees

A spanning tree of a graph G is a subgraph that is a tree containing all vertices of G



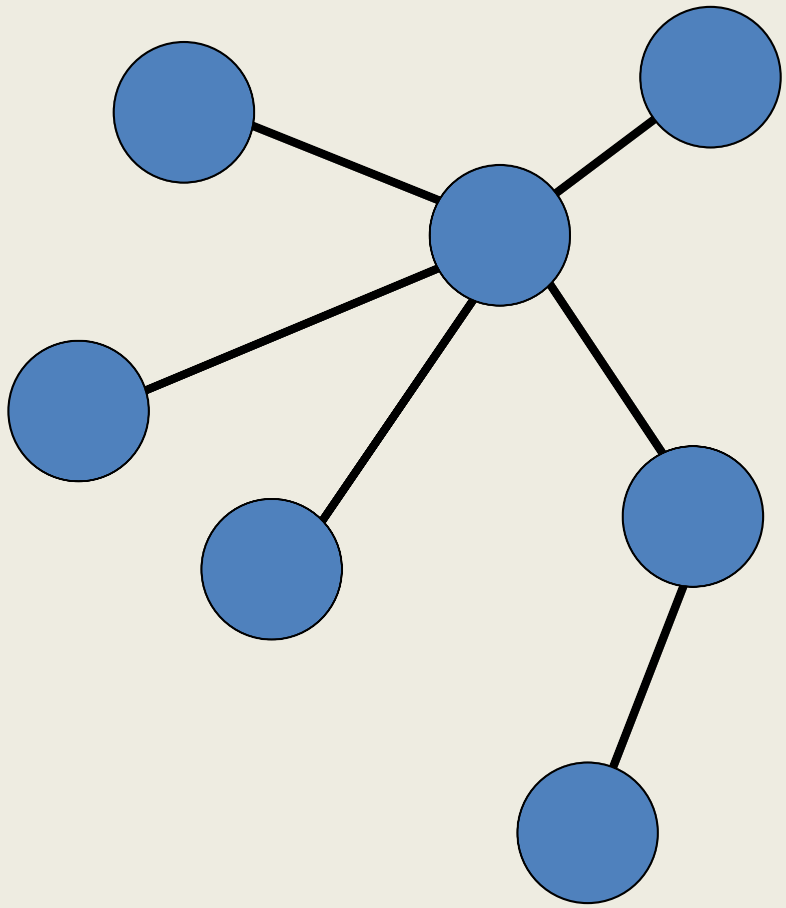
Spanning trees

A spanning tree of a graph G is a subgraph that is a tree containing all vertices of G



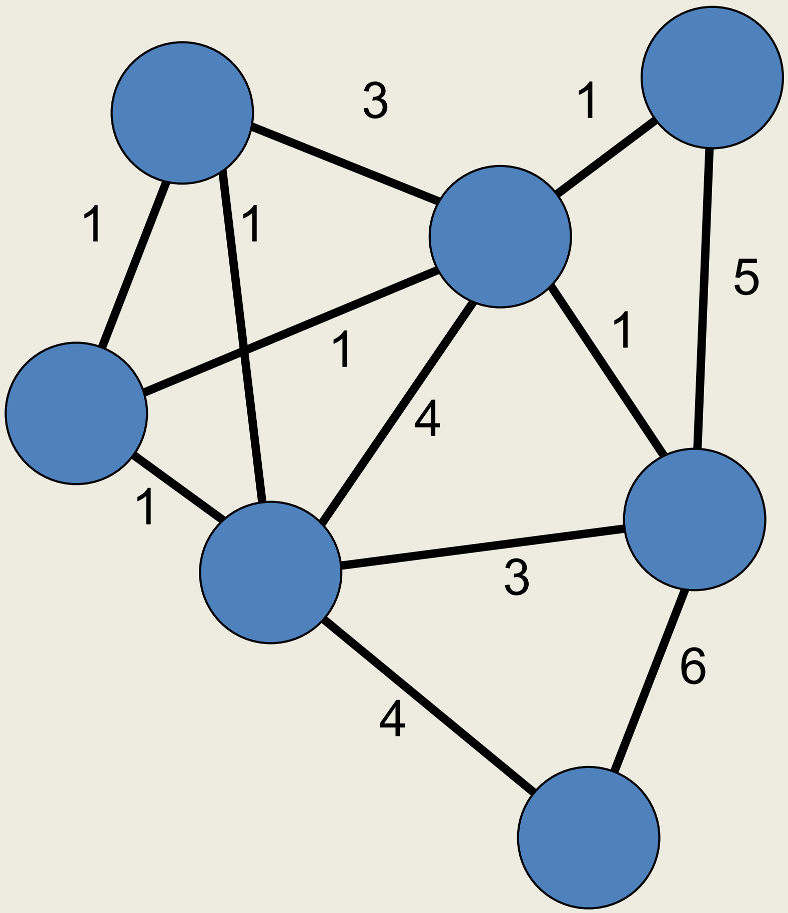
Spanning trees

A spanning tree of a graph G is a subgraph that is a tree containing all vertices of G



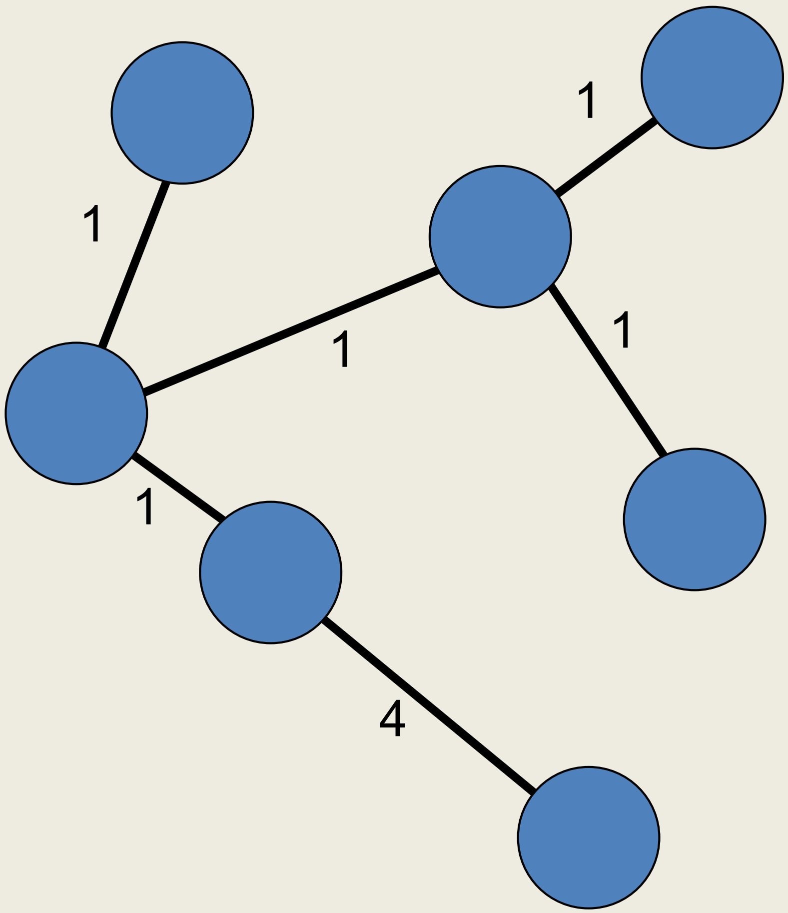
Minimal spanning trees

A minimal spanning tree of a weighted graph G is a spanning tree of minimum weight



Minimal spanning trees

A minimal spanning tree of a weighted graph G is a spanning tree of minimum weight



Kruskal's algorithm

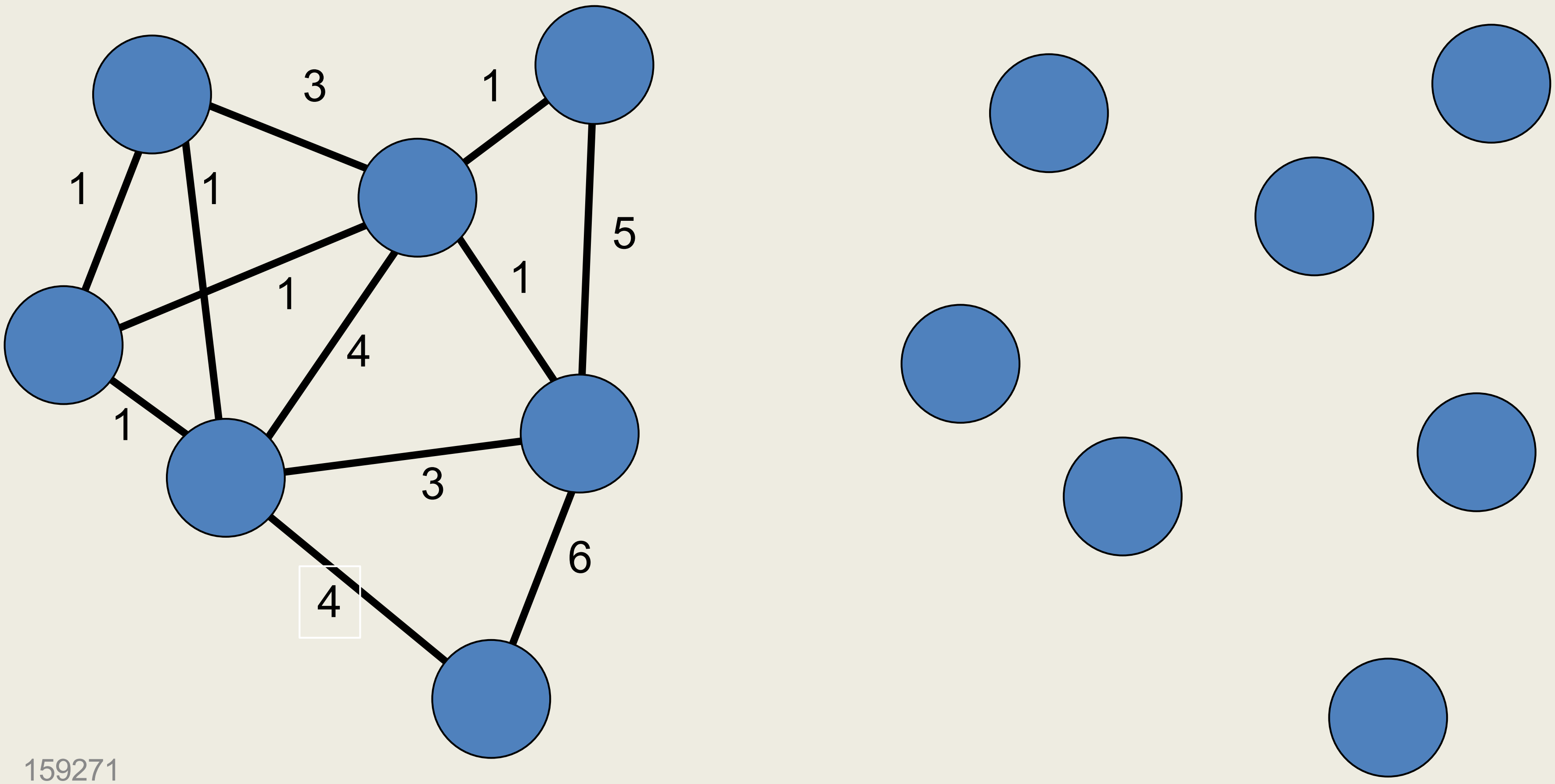
To find a minimal spanning tree of G:

start with all the vertices and no edges

greedy rule: add an edge of minimum weight that does not make a cycle

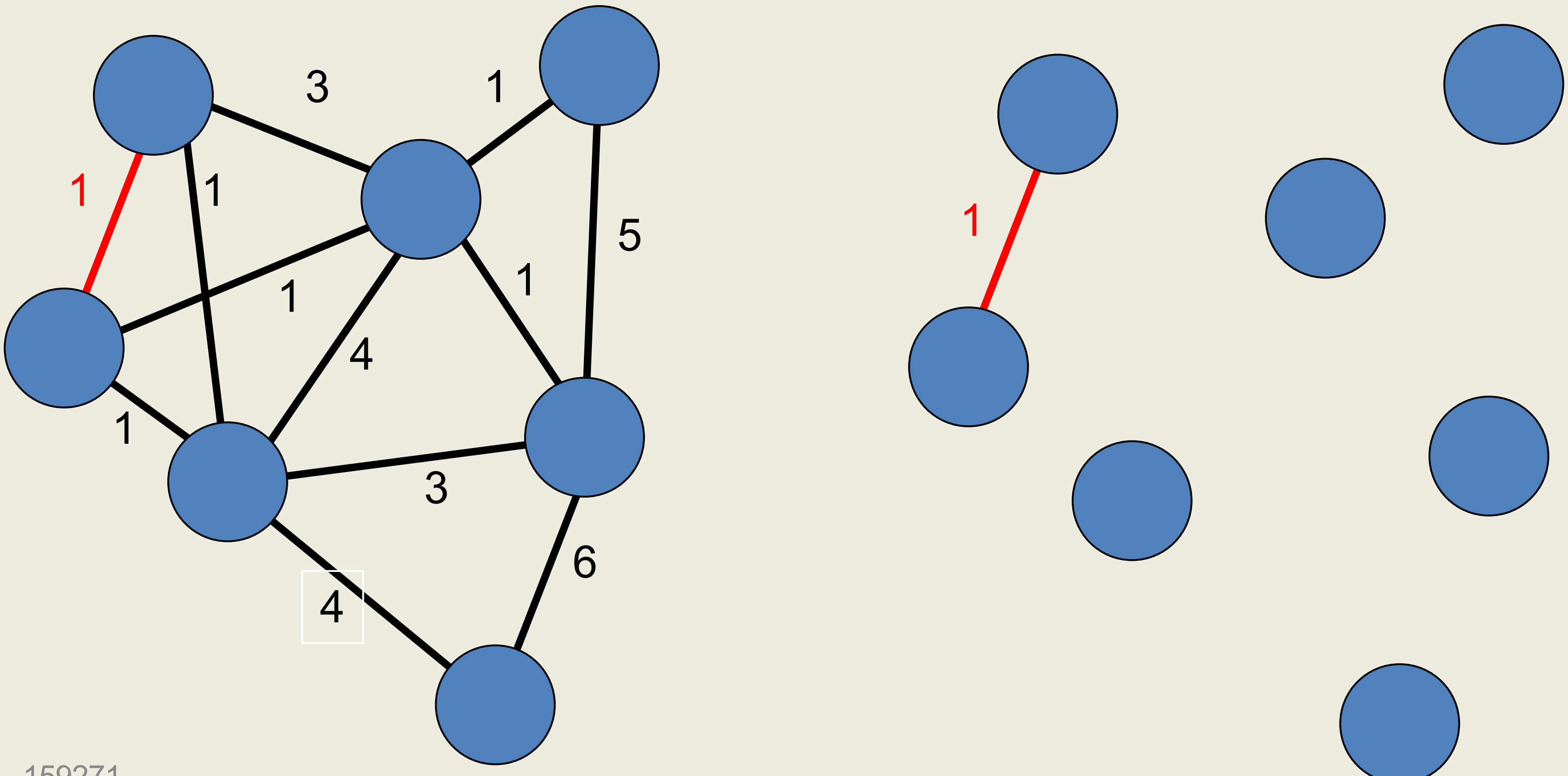
Kruskal’s algorithm

greedy rule: add an edge of minimum weight that does not make a cycle



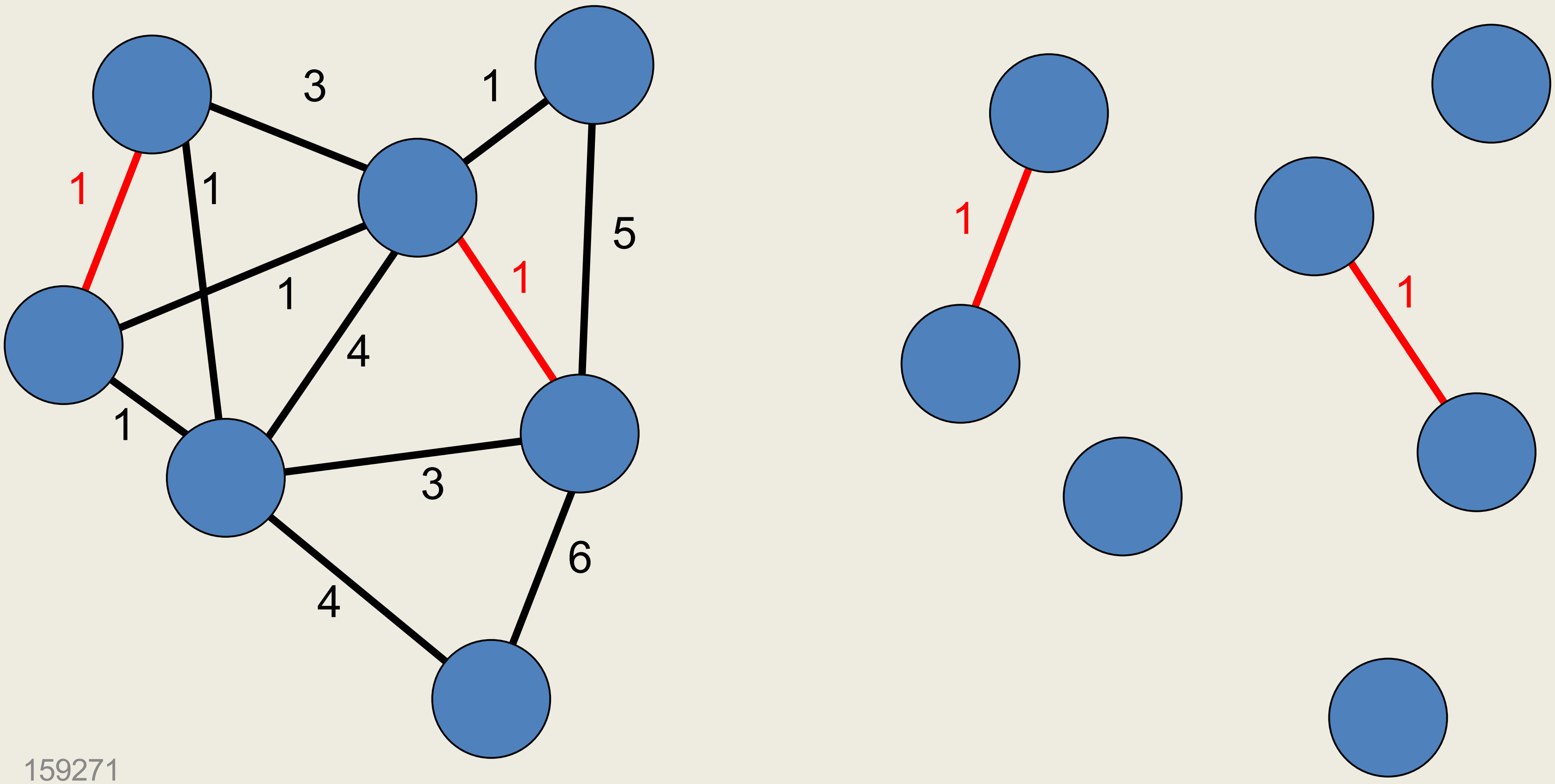
Kruskal's algorithm

greedy rule: add an edge of minimum weight that does not make a cycle



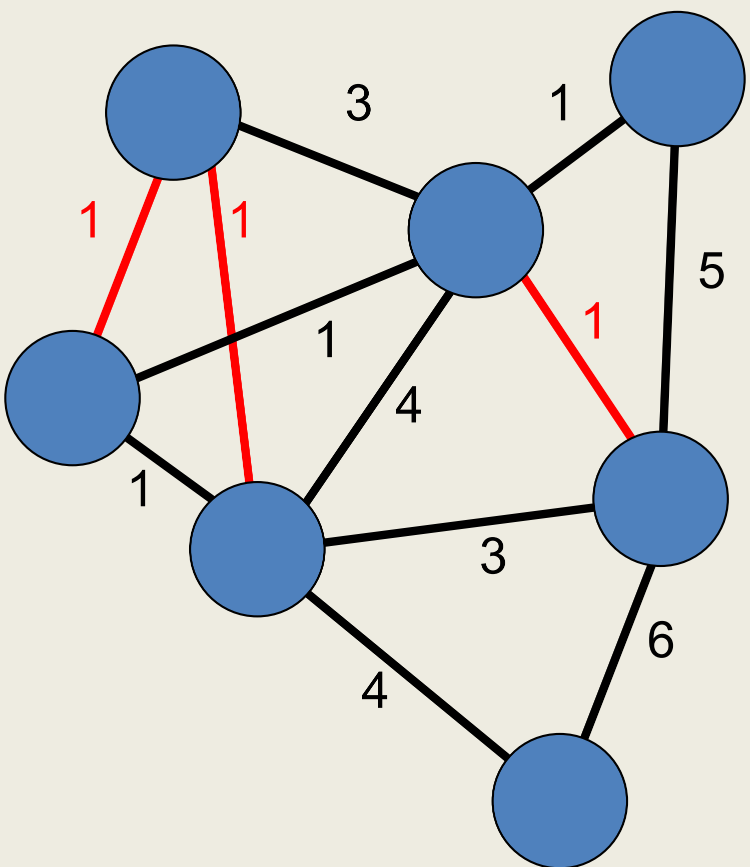
Kruskal’s algorithm

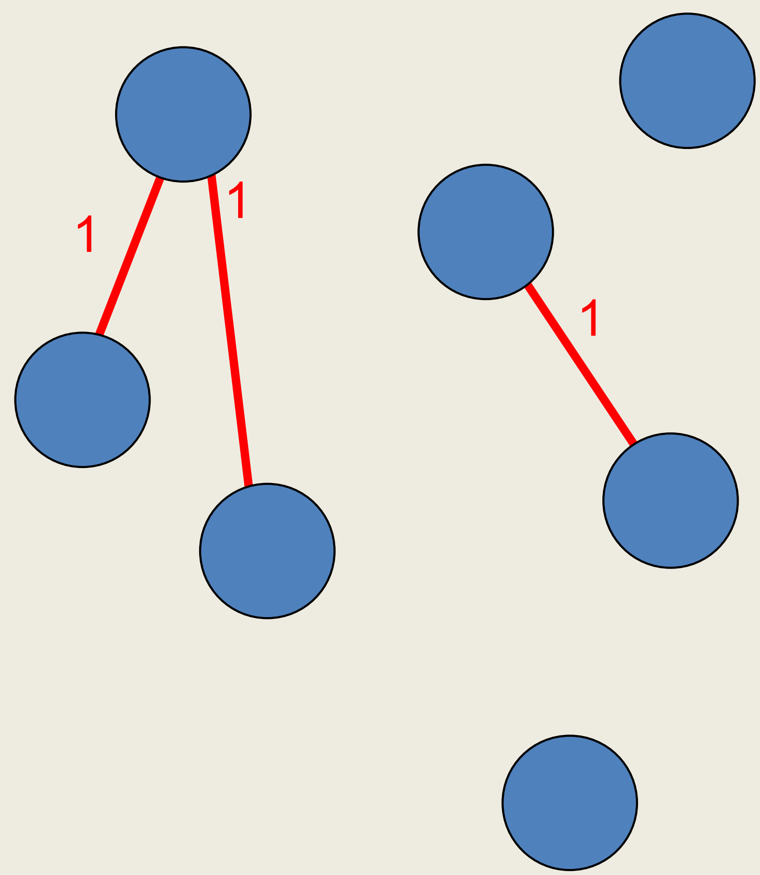
greedy rule: add an edge of minimum weight that does not make a cycle



Kruskal’s algorithm

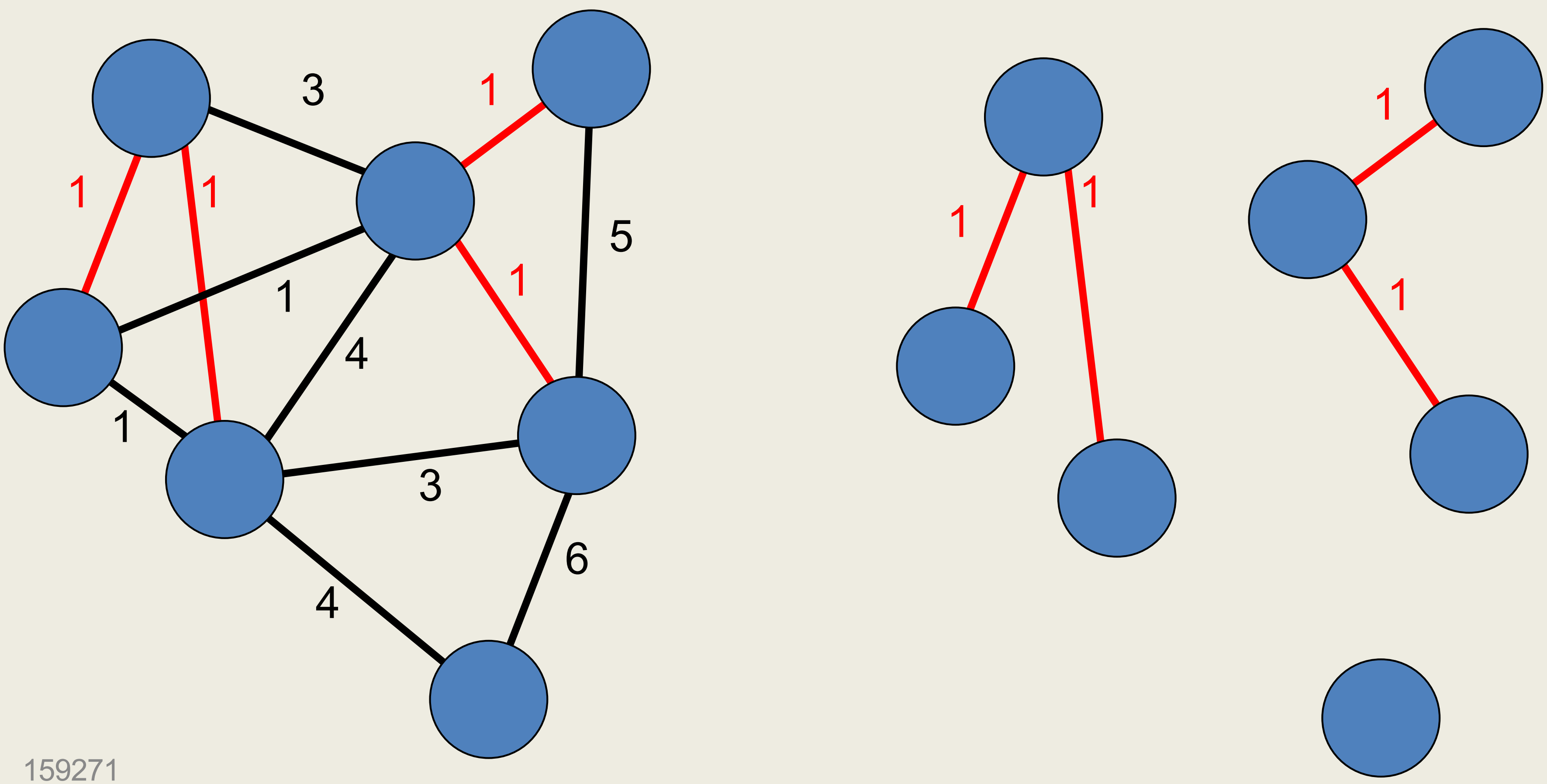
greedy rule: add an edge of minimum weight that does not make a cycle





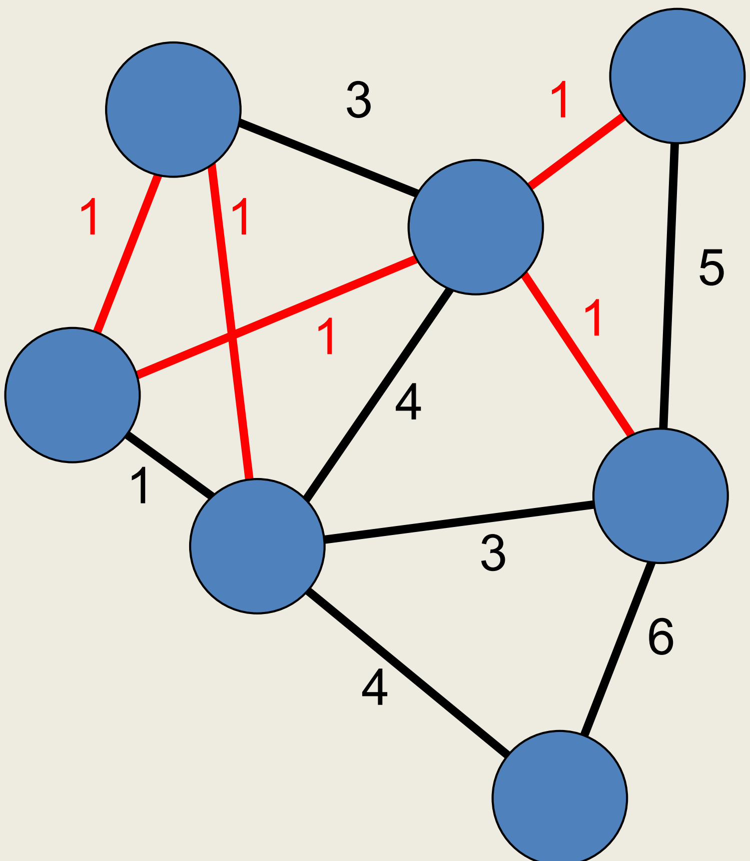
Kruskal's algorithm

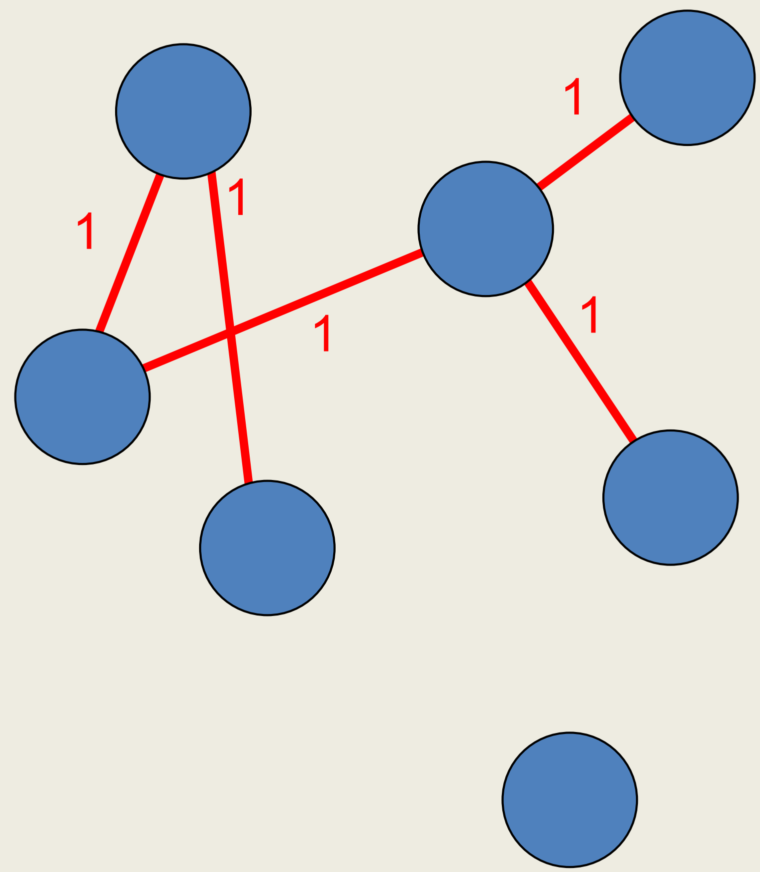
greedy rule: add an edge of minimum weight that does not make a cycle



Kruskal's algorithm

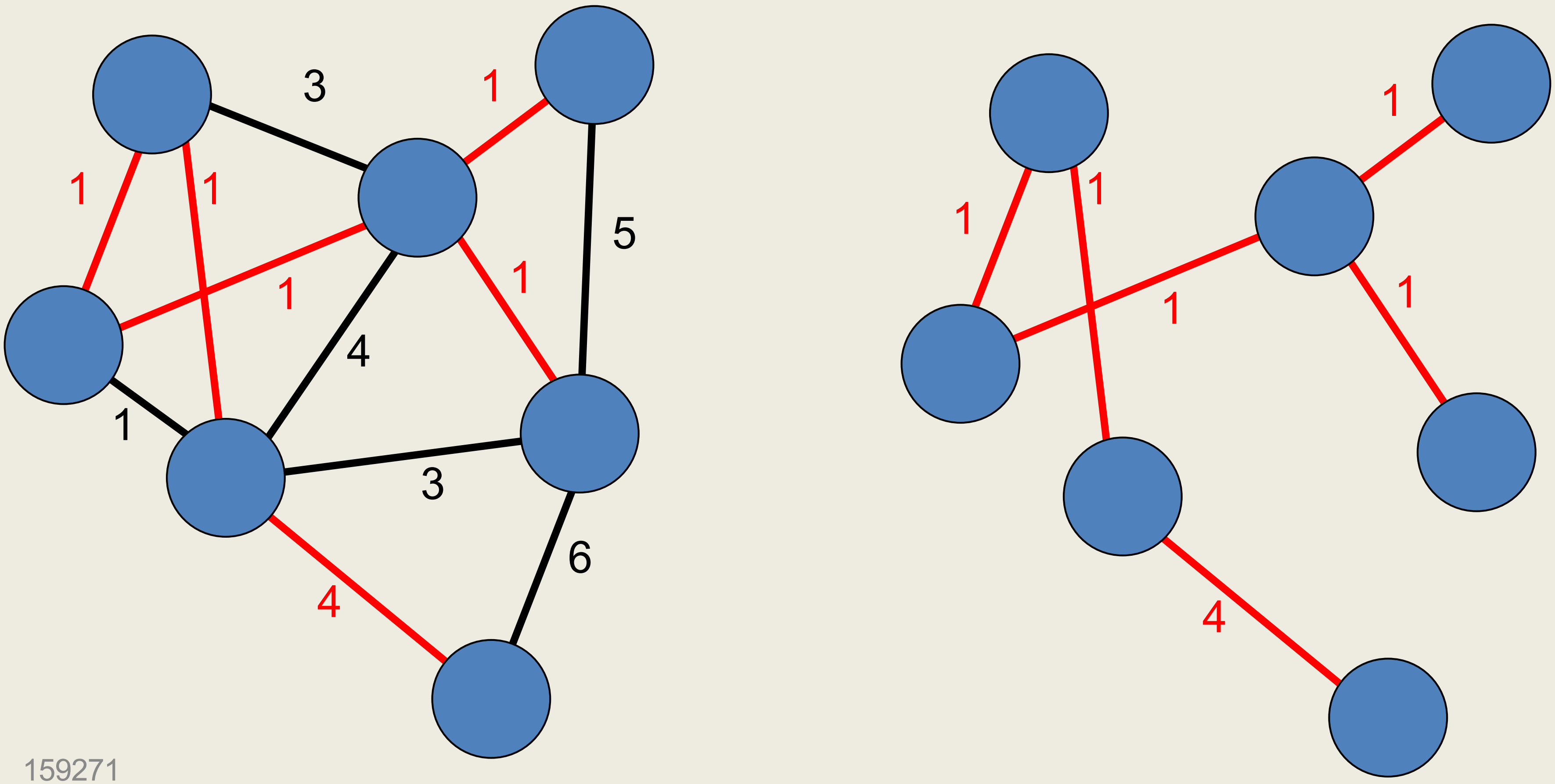
greedy rule: add an edge of minimum weight
that does not make a cycle





Kruskal’s algorithm

greedy rule: add an edge of minimum weight that does not make a cycle



Prim's algorithm

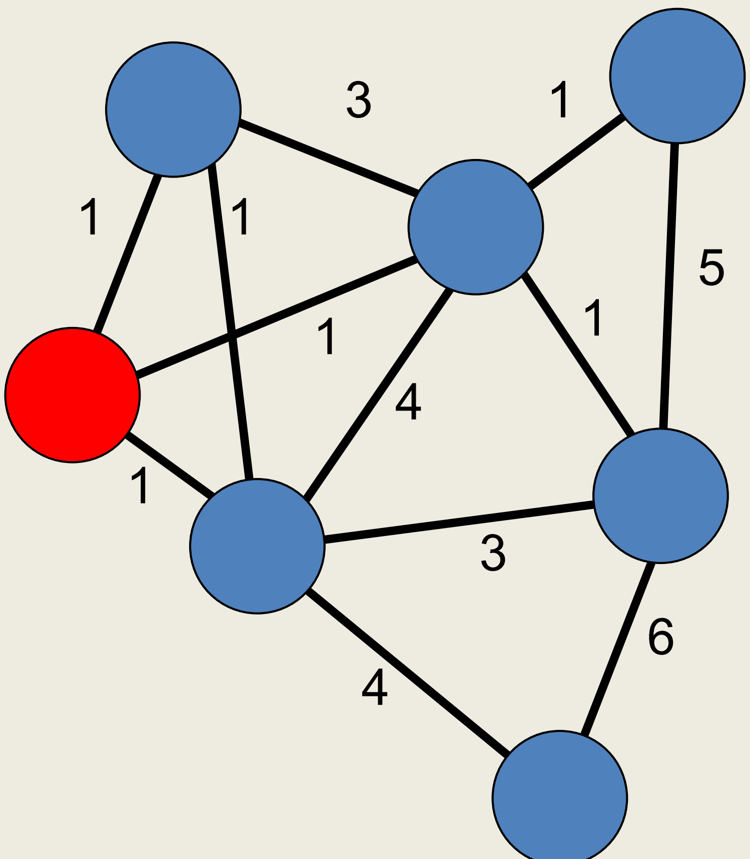
To find a minimal spanning tree of G:

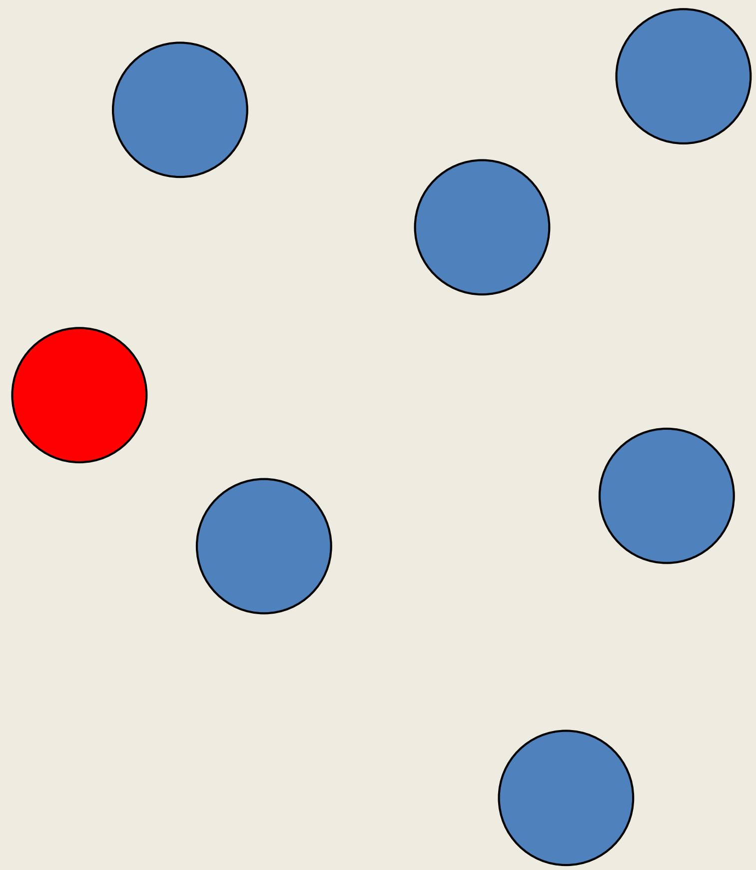
start with a start vertex and no edges

greedy rule: add an edge of minimum weight that has one endpoint in current tree and other endpoint not in current tree

Prim’s algorithm

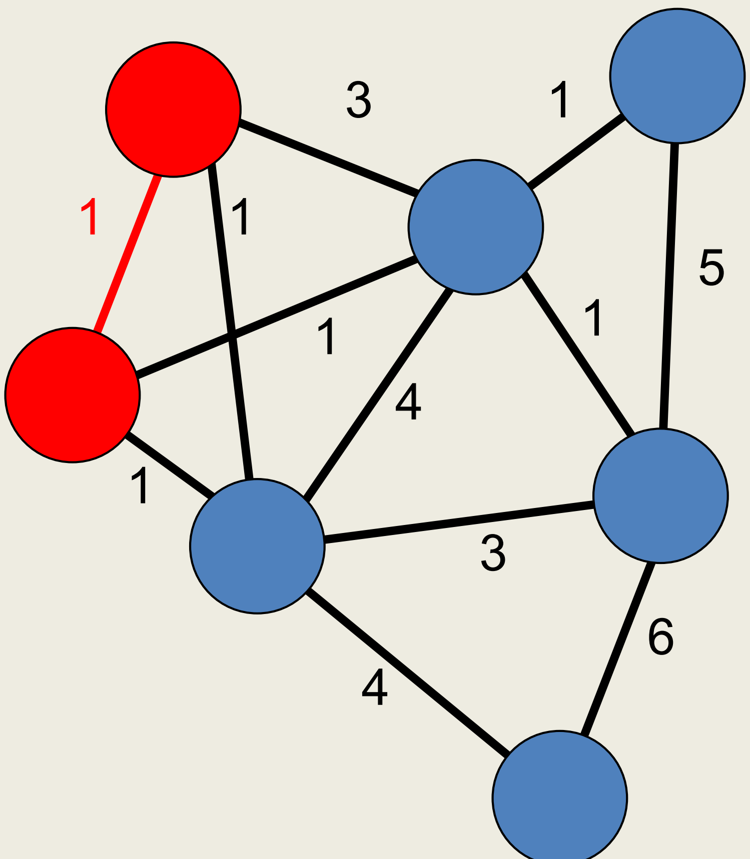
greedy rule: add an edge of minimum weight that has one endpoint in current tree and other endpoint not in current tree

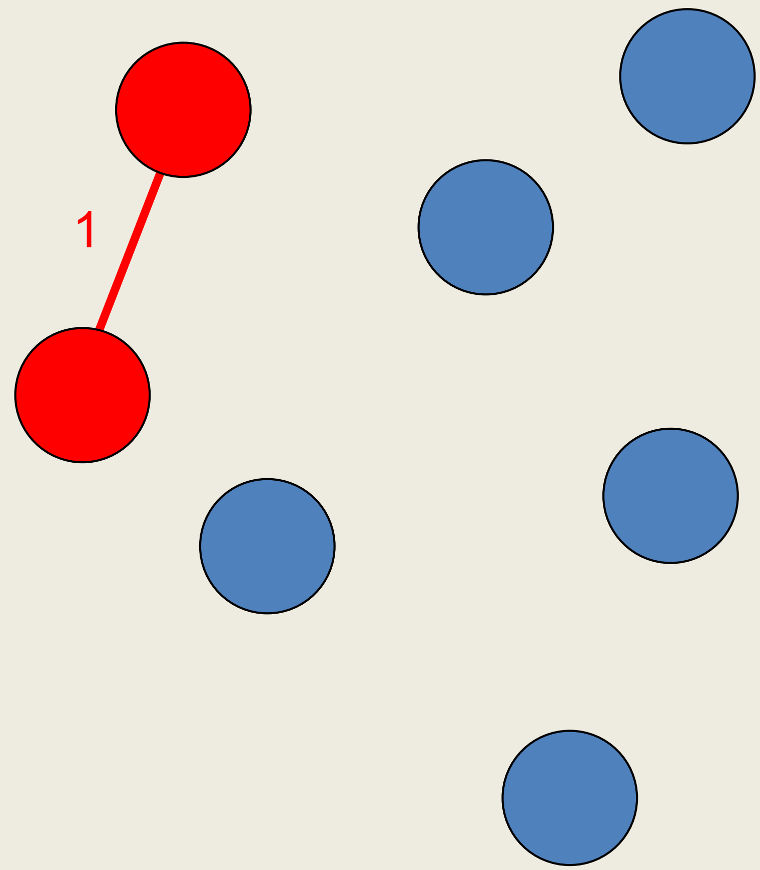




Prim’s algorithm

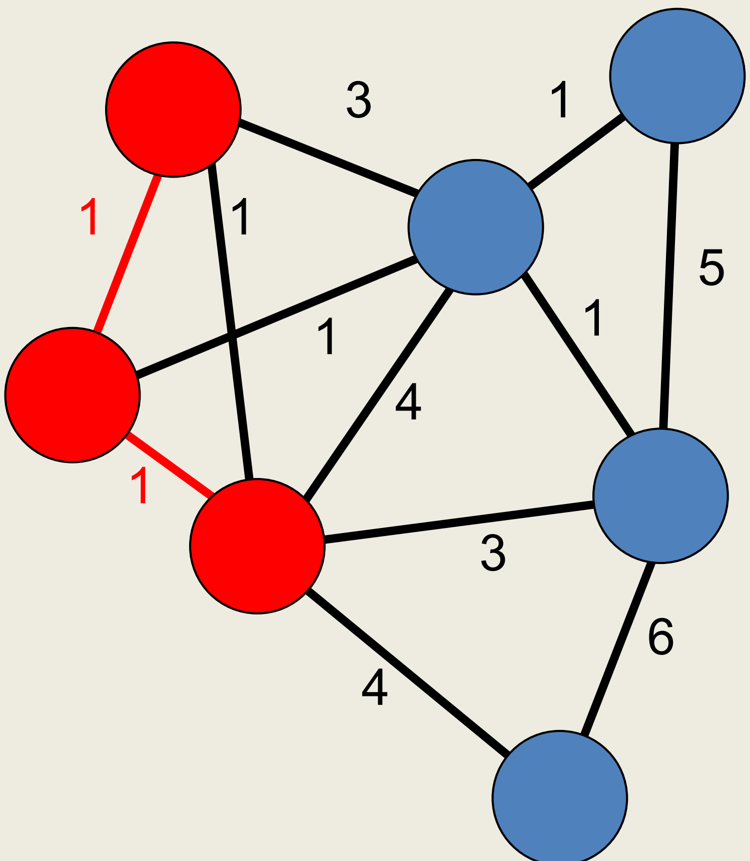
greedy rule: add an edge of minimum weight that has one endpoint in current tree and other endpoint not in current tree

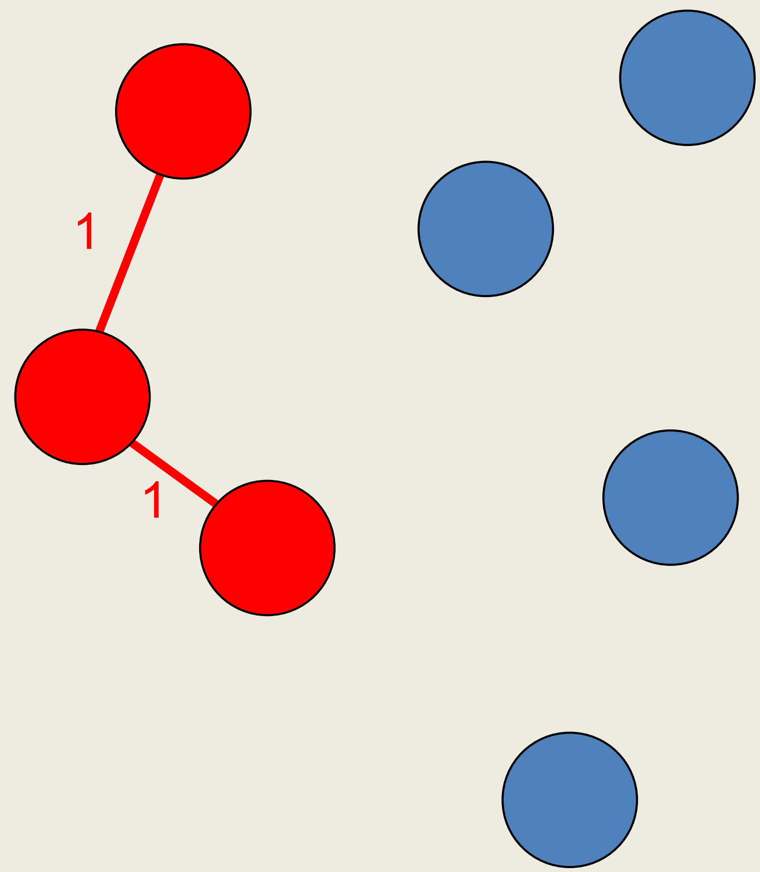




Prim’s algorithm

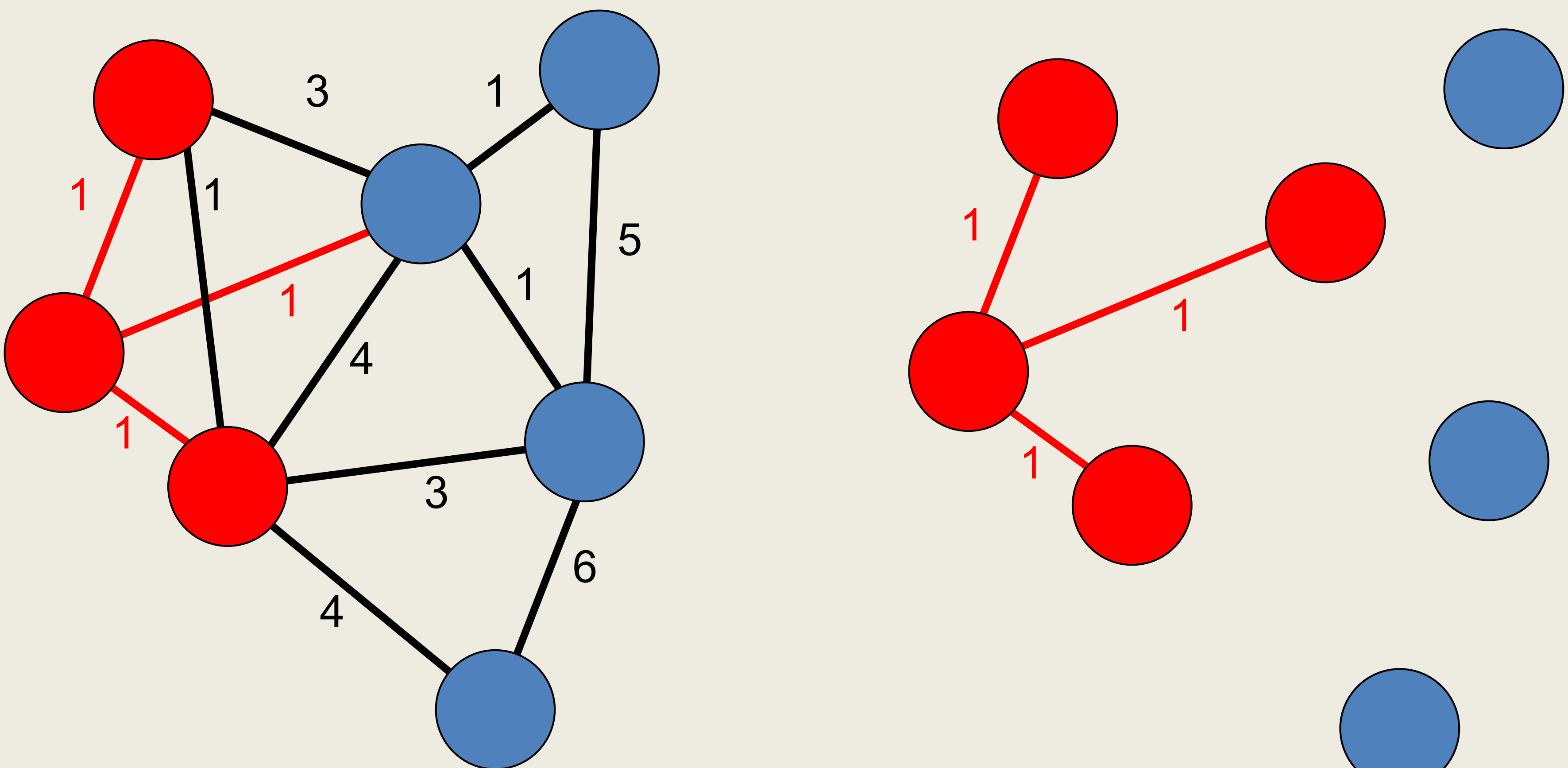
greedy rule: add an edge of minimum weight that does not make a cycle





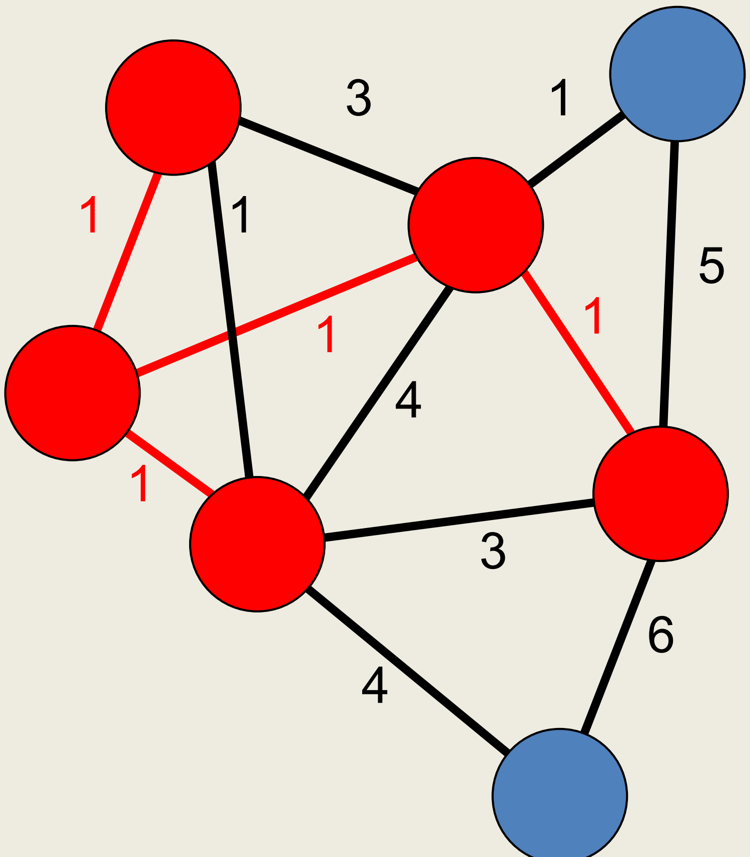
Prim's algorithm

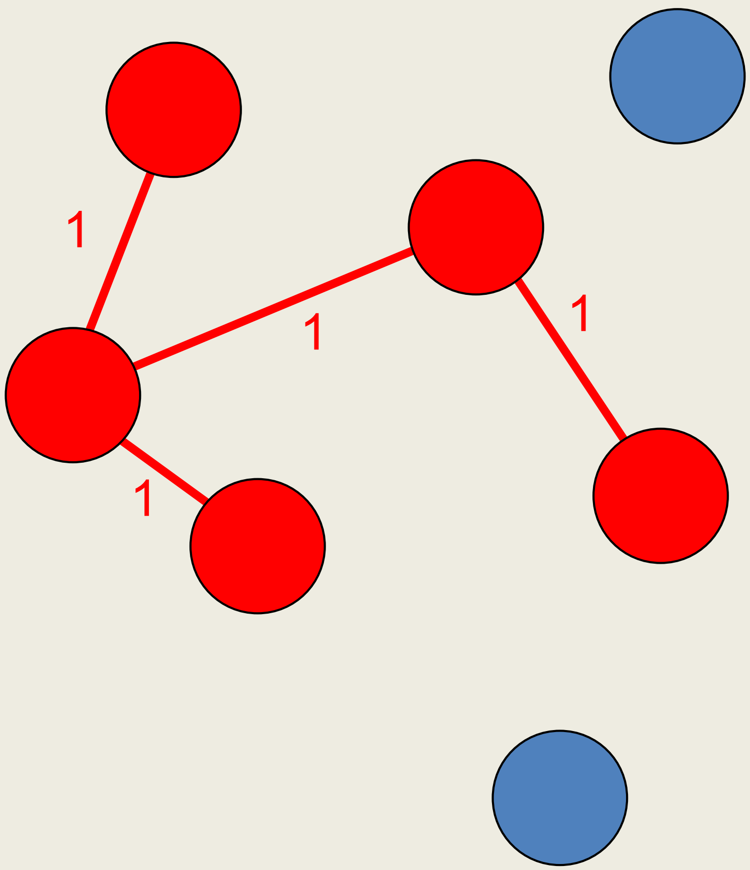
greedy rule: add an edge of minimum weight that does not make a cycle



Prim's algorithm

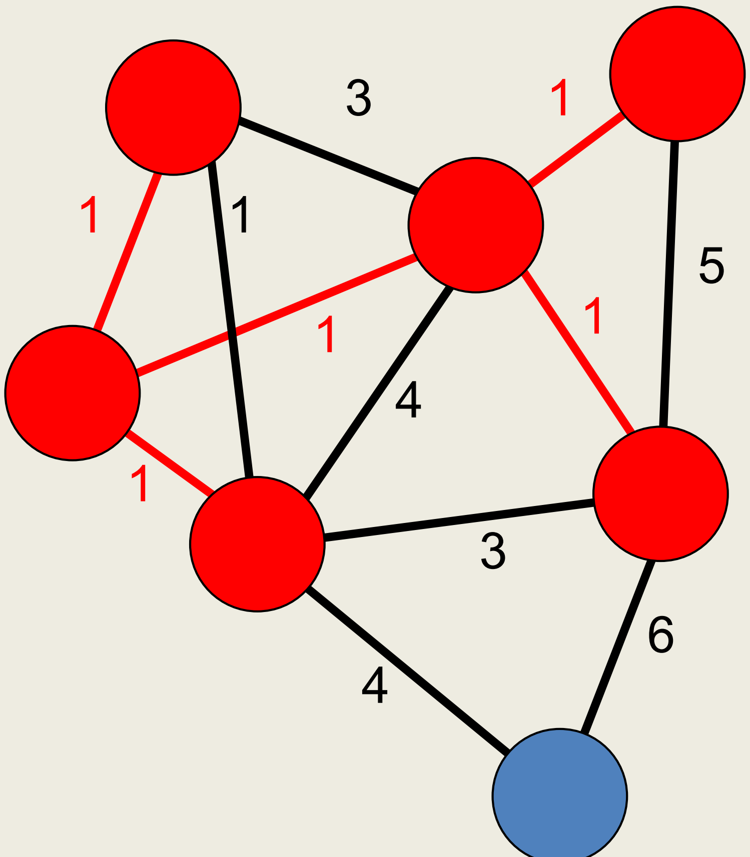
greedy rule: add an edge of minimum weight that does not make a cycle

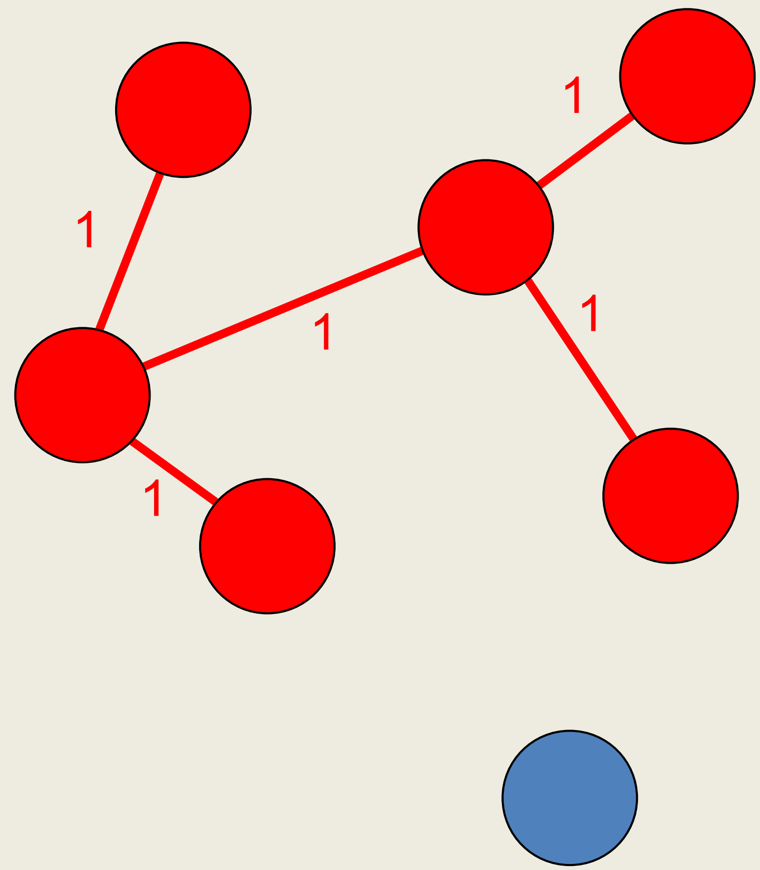




Prim's algorithm

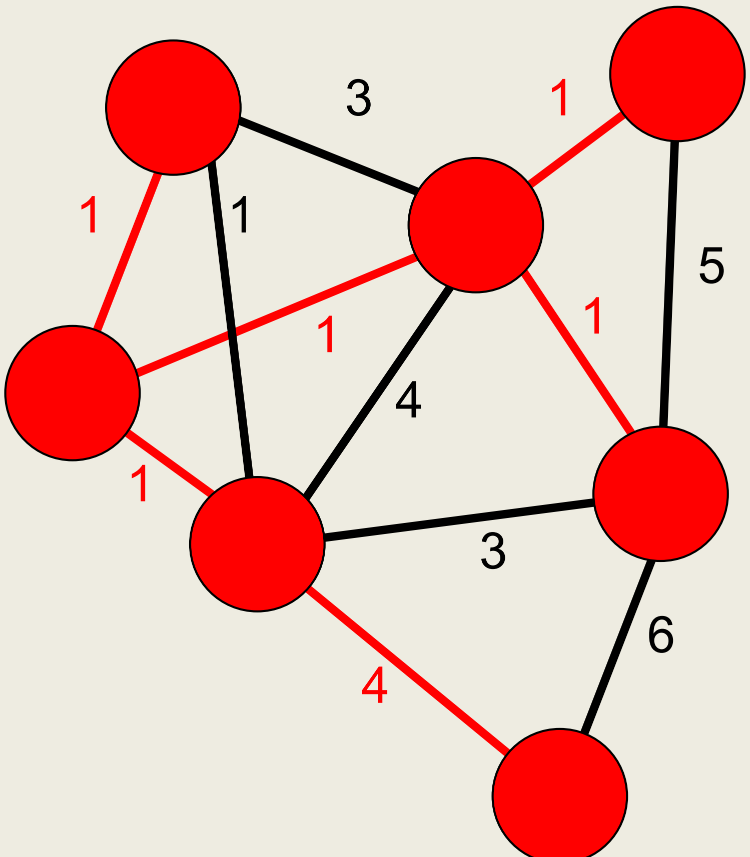
greedy rule: add an edge of minimum weight that does not make a cycle

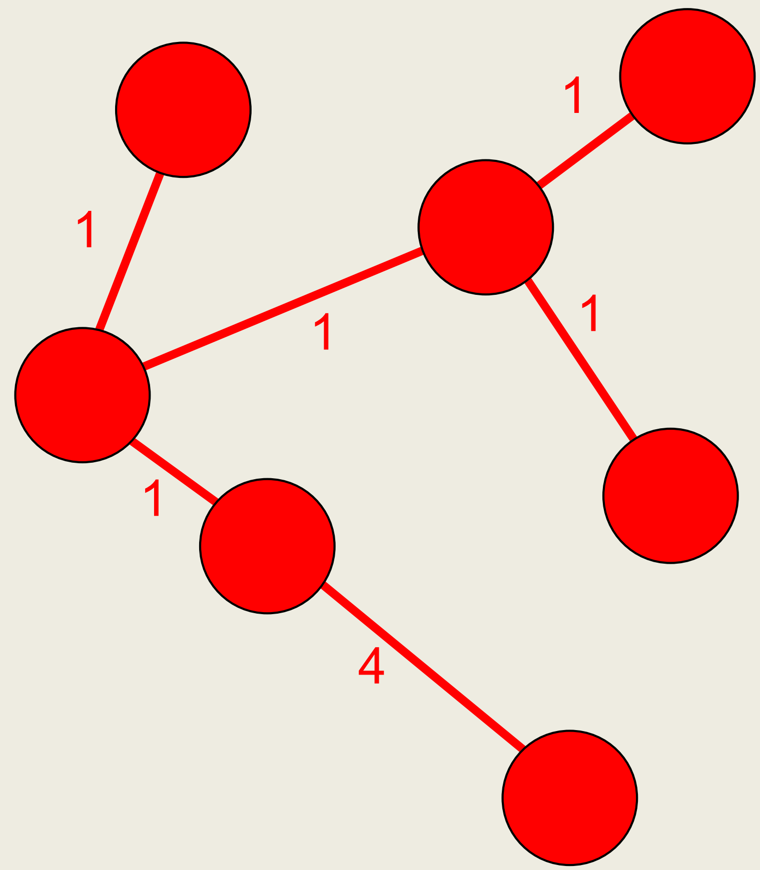




Prim’s algorithm

greedy rule: add an edge of minimum weight that does not make a cycle





Kruskal’s algorithm - correctness

Lemma:

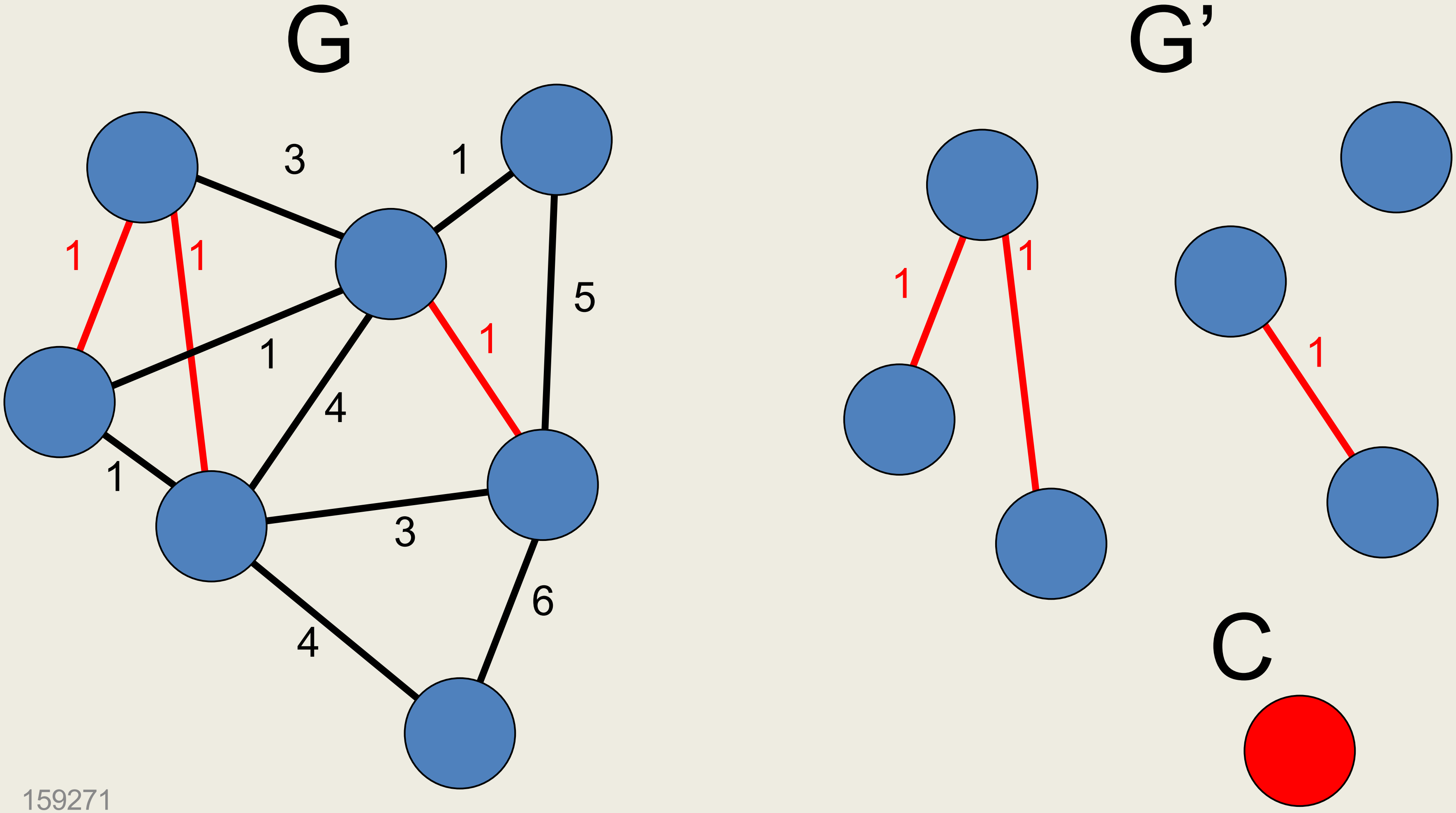
Let G be be a connected weighted graph and let G’ be a subgraph of a minimal spanning tree of G.

Let C be a component of G’ and let S be the set of edges with one endpoint in C, the other not in C.

If we add a minimum weight edge in S to G’ then the resulting graph is also contained in a minimal spanning tree of G.

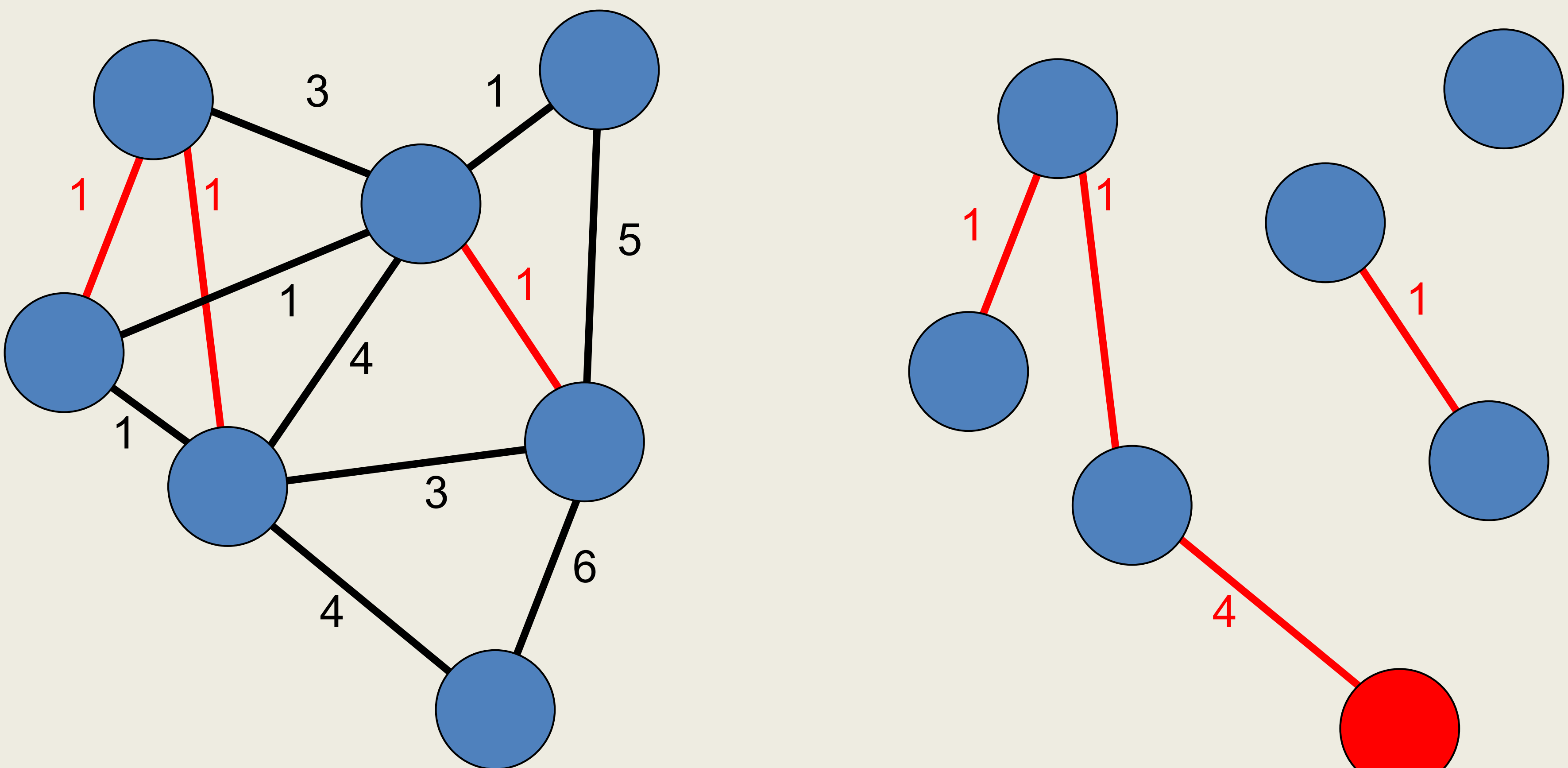
Kruskal's algorithm

Choose any component C



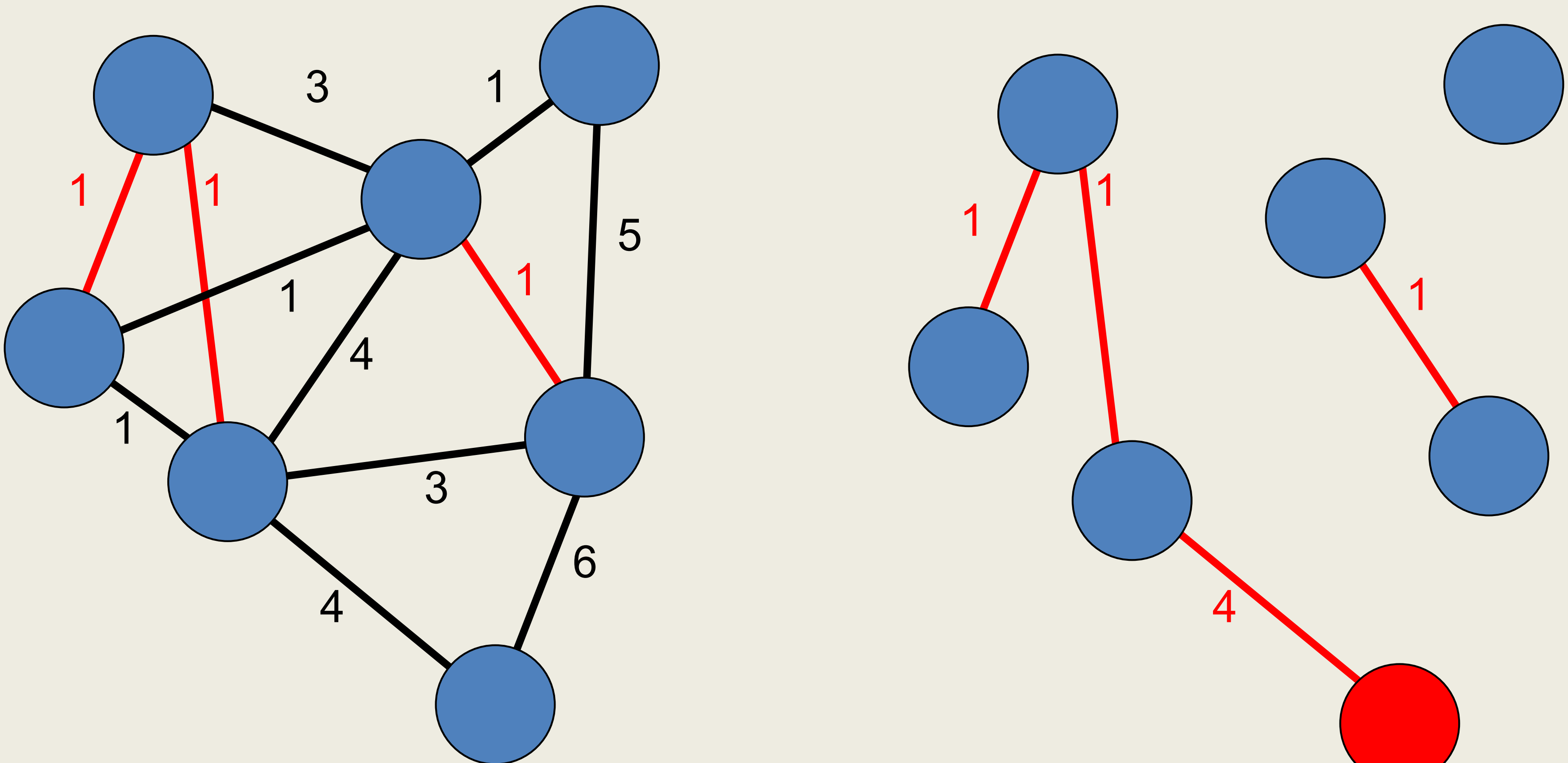
Kruskal’s algorithm

Choose any component C, add minimum weight edge C -> T\C



Kruskal's algorithm

resulting graph is also contained in a minimal spanning tree of G



Kruskal's algorithm - correctness

Use lemma and induction:

At each step the subgraph constructed is contained in a minimal spanning tree of G.

1. Step 1, subgraph with no edges is contained in every minimal spanning tree of G
2. Step k, assume that the current subgraph is contained in a minimal spanning tree of G.
3. Let (v,w) be the next edge chosen, let C be the component to which v belongs. Edge (v,w) is minimum weight edge with one endpoint in C, the other not (since it is minimum weight from any component to any other).
4. By lemma, subgraph at Step k+1 is contained in a minimal spanning tree of G.

Kruskal's algorithm - correctness

Proof of lemma:

Let G' be a subgraph of minimal spanning tree T.

Let (v,w) be a minimum weight edge chosen with one endpoint in C.

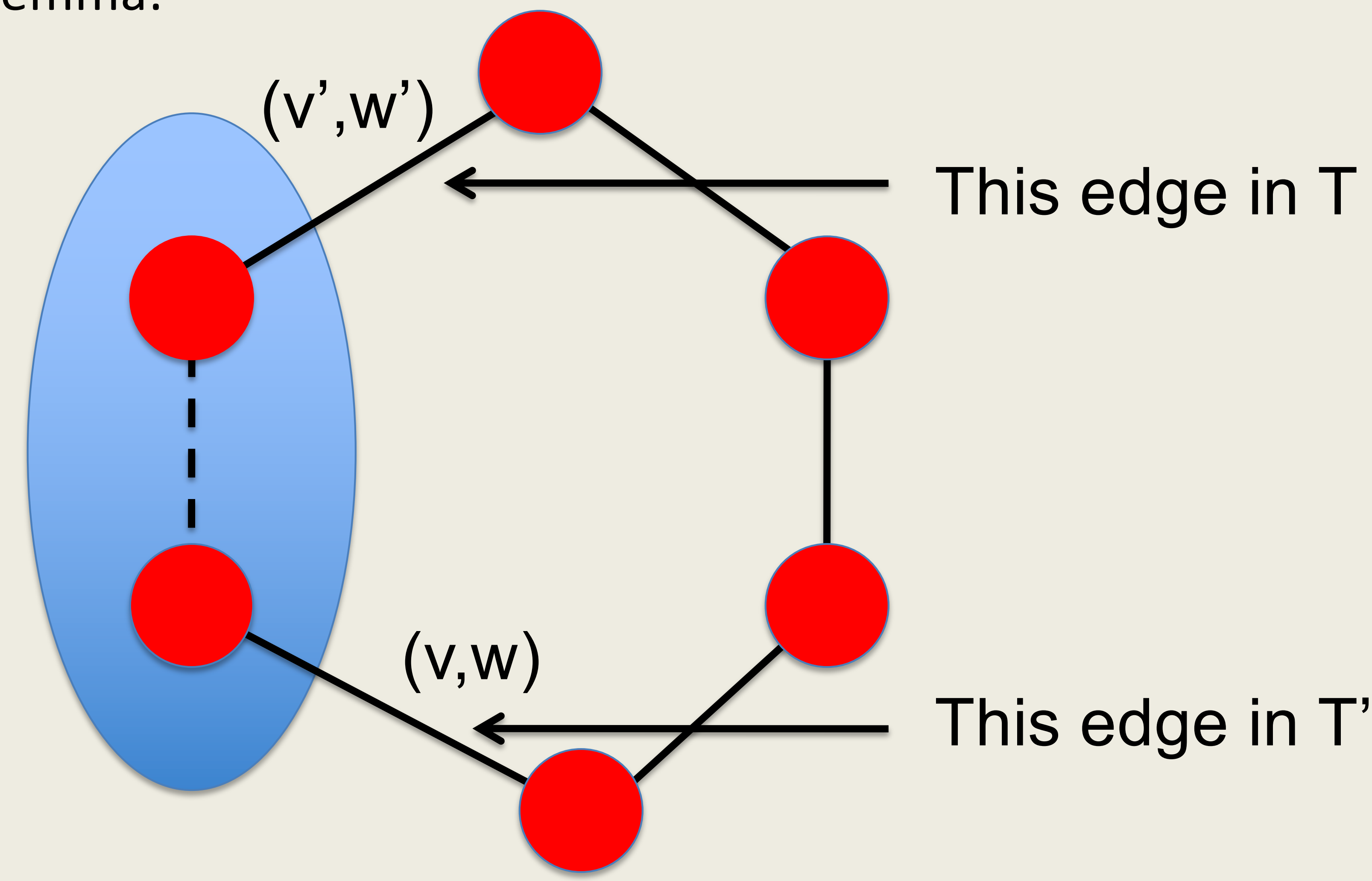
If (v,w) in T ,then we are done.

Suppose T doesn't contain (v,w): add (v,w) to T, remove an edge from cycle S created in T to get T' another spanning tree for G.

Choose edge (v'w') to remove as first edge we get to on the cycle that "leaves" C.

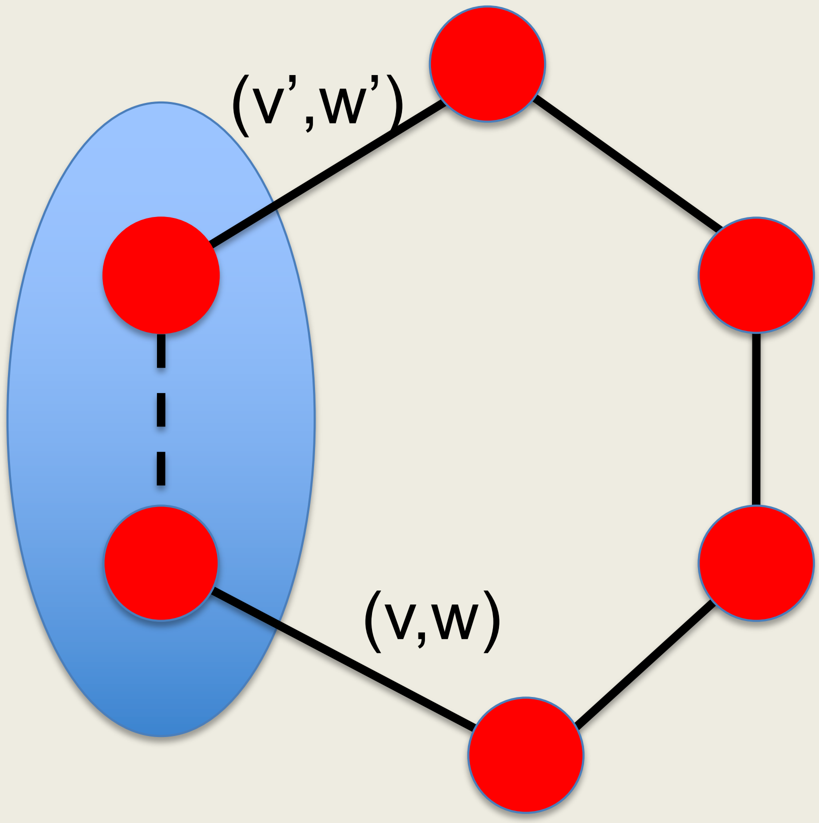
Kruskal's algorithm - correctness

Proof of lemma:



Kruskal's algorithm - correctness

Proof of lemma:

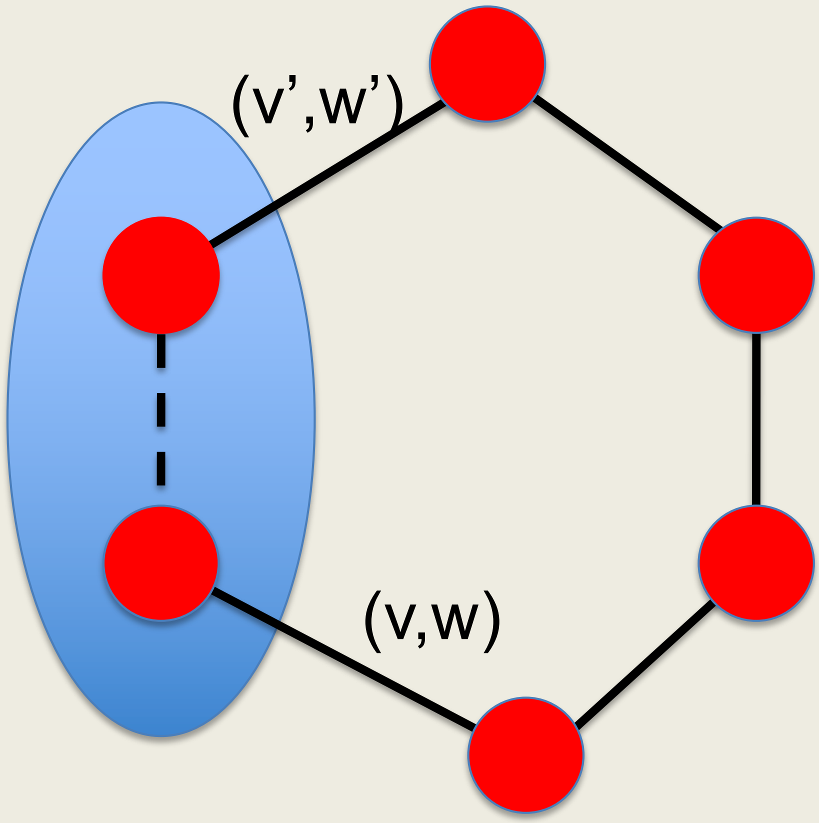


Weight of (v',w') is at least weight of (v,w), since (v,w) minimum weight edge leaving C.

=> T must have weight at least weight of T'.

Kruskal's algorithm - correctness

Proof of lemma:



T must have weight at least T'.

⇒ Since T is minimal spanning tree, T and T' have the same weight.

⇒ T' a minimal spanning tree containing G' + (v,w)

Kruskal's algorithm - implementation

Want to select the edges in non-decreasing order of weight.

=> represent the graph as a list of edges and their weights, sorted by weight.

Want to determine if adding an edge creates a cycle.

=> keep track of connected components = sets of vertices connected together

Kruskal’s algorithm - implementation

```
graph = {
        'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
        'edges': set([
            (1, 'A', 'B'),
            (5, 'A', 'C'),
            (3, 'A', 'D'),
            (4, 'B', 'C'),
            (2, 'B', 'D'),
            (1, 'C', 'D'),
            ])
        }
```

Kruskal's algorithm - implementation

```
def kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)

    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, vertice1, vertice2 = edge
        if find(vertice1) != find(vertice2):
            union(vertice1, vertice2)
            minimum_spanning_tree.add(edge)
    return minimum_spanning_tree
```

Kruskal’s algorithm - implementation

```python
parent = dict()
rank = dict()

def make_set(vertice):
    parent[vertice] = vertice
    rank[vertice] = 0

def find(vertice):
    if parent[vertice] != vertice:
        parent[vertice] = find(parent[vertice])
    return parent[vertice]

def union(vertice1, vertice2):
    root1 = find(vertice1)
    root2 = find(vertice2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            parent[root2] = root1
        else:
            parent[root1] = root2
            if rank[root1] == rank[root2]: rank[root2] += 1
```

Prim’s algorithm - correctness

Use lemma and induction:

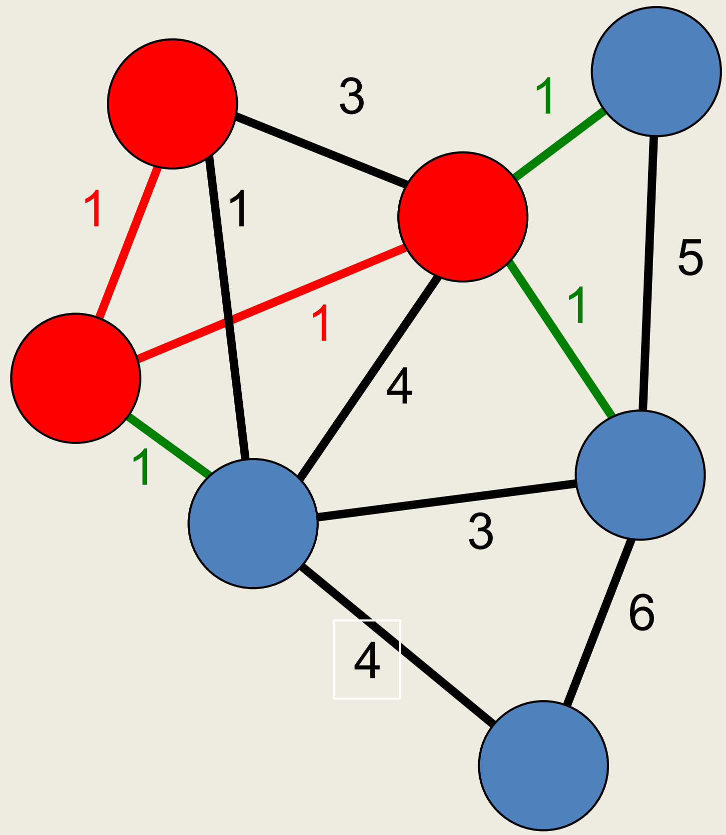
At each step the tree T constructed is contained in a minimal spanning tree of G.

1. Step 1, tree with no edges is contained in every minimal spanning tree of G
2. Step k, assume that the current tree is contained in a minimal spanning tree of G.
3. Let (v,w) be the next edge chosen, where v is in T and w not in T. Let G’ be T together with all vertices not in T. Then T is a component of G’ and (v,w) minimum weight edge with one endpoint in T, one not.
4. By lemma, T at Step k+1 is contained in a minimal spanning tree of G.

Prim's algorithm - implementation

Want to keep track of candidate edges to add to the current tree.

retain just one minimum weight edge from each non-tree vertex to current tree



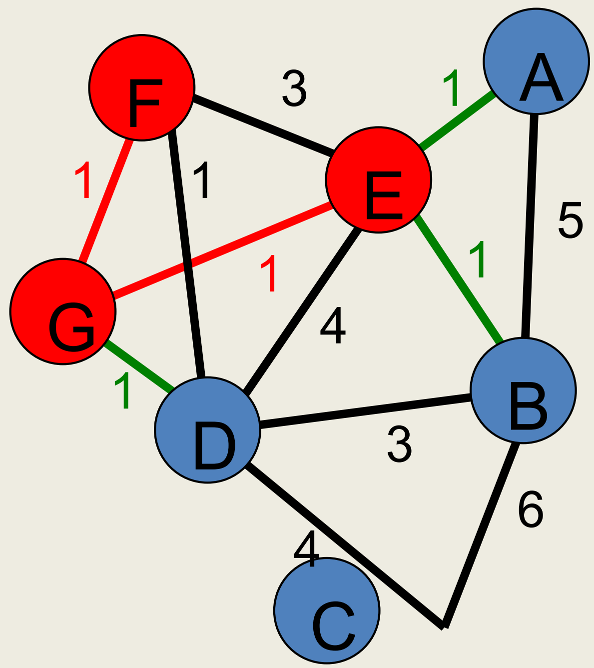
Prim’s algorithm - implementation

Want to keep track of candidate edges to add to the current tree.

retain just one minimum weight edge from each non-tree vertex to current tree

keep a list of non-tree vertices with edge weight and parent in the tree recorded

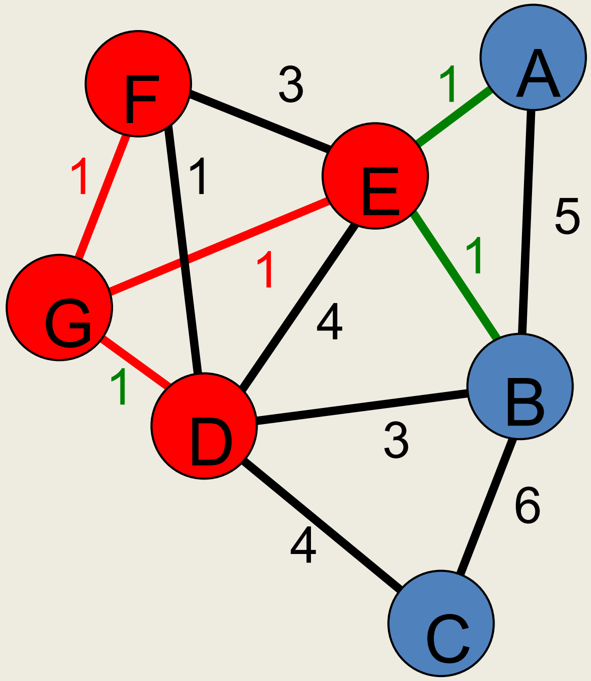
Prim’s algorithm - implementation



vertex	min weight	parent
A	1	E
B	1	E
C	∞	-
D	1	G

Prim’s algorithm - implementation

After each addition to tree update weights and parents



vertex	min weight	parent
A	1	E
B	1	E
C	4	D
~~D~~	~~1~~	~~G~~

Prim’s algorithm - implementation

- represent the graph using adjacency lists
- store non-tree vertices paired with edges weights in a heap (binary min-heap)
- makes it efficient to find the lowest weight edge at each step, delete vertex from heap and add to tree
- store parent values in an array (list)
- parent values (for adjacent vertices only) need to be updated after each addition to tree
- parent array used to construct tree as output

Prim’s algorithm - implementation

`h` is an abstract data type supporting:

`h.init(key,n)` – initialises h to values in key

`h.del()` – deletes item in h with smallest weight returns corresponding vertex

`h.isin(w)` – returns true if vertex w is in h, false otherwise

`h.keyval(w)` - returns the weight corresponding to vertex w

`h.decrease(w, wgt)` – changes the weight corresponding to w to wgt

Prim's algorithm - implementation

graph is represented using adjacency lists

`adj[i]` –

reference to first node in a linked list representing vertices adjacent to `i`

each node in the list has members –

`ver` (the vertex), `weight` (weight of edge `(i, ver)`) and `next` (reference to next node in linked list)

start vertex is `start`

parent of vertex `i` ≠ start is `parent[i]`

`parent[start] = 0`

Prim’s algorithm - implementation

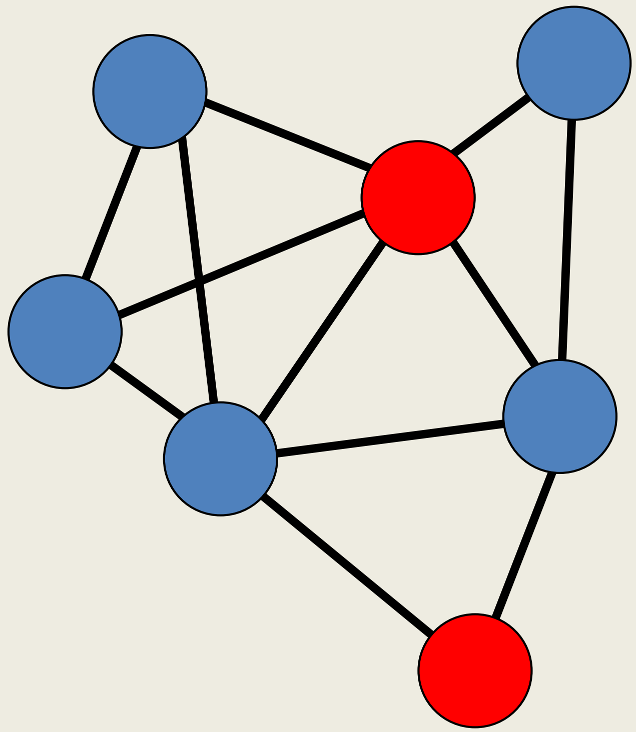
```
prim(adj, start, parent):
   n = adj.size
   for i = 1 to n:
    key[i] = ∞
   key[start] = 0
   parent [start] = 0
   h.init(key,n)
   for i = 1 to n:
    v = h.del()
    ref = adj[v]
    while ref != null:
        w = ref.ver
        if h.isin(w) && ref.weight < h.keyval(w) :
            parent[w] = v
            h.decrease(w, ref.weight)
        ref = ref.next
```

Prim's algorithm - implementation

```
prim(adj, start, parent):
   n = adj.size
   for i = 1 to n:
    key[i] = ∞
   key[start] = 0
   parent [start] = 0
   h.init(key,n)
   for i = 1 to n:
    v = h.del()
    ref = adj[v]
    while ref != null:
        w = ref.ver
        if h.isin(w) && ref.weight < h.keyval(w) :
            parent[w] = v
            h.decrease(w, ref.weight)
        ref = ref.next
```

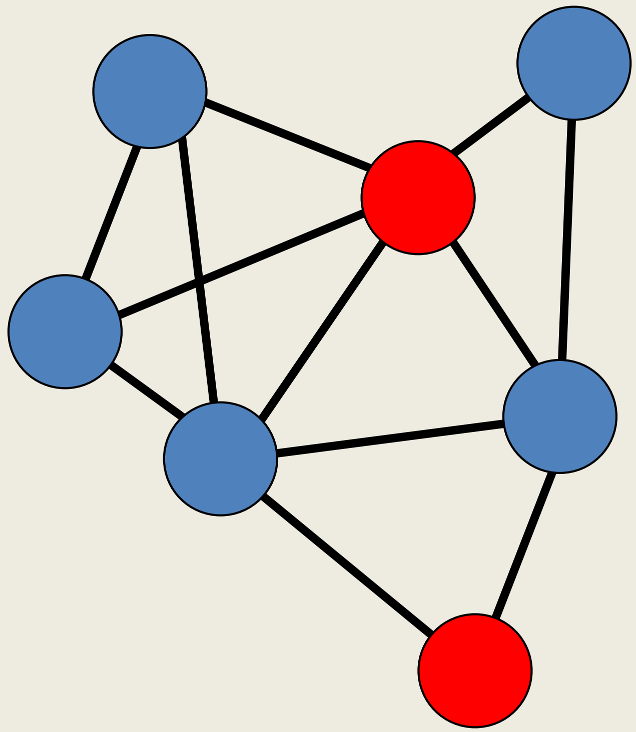
exercise : implement these abstract data types and the algorithm in Python

Dominating set



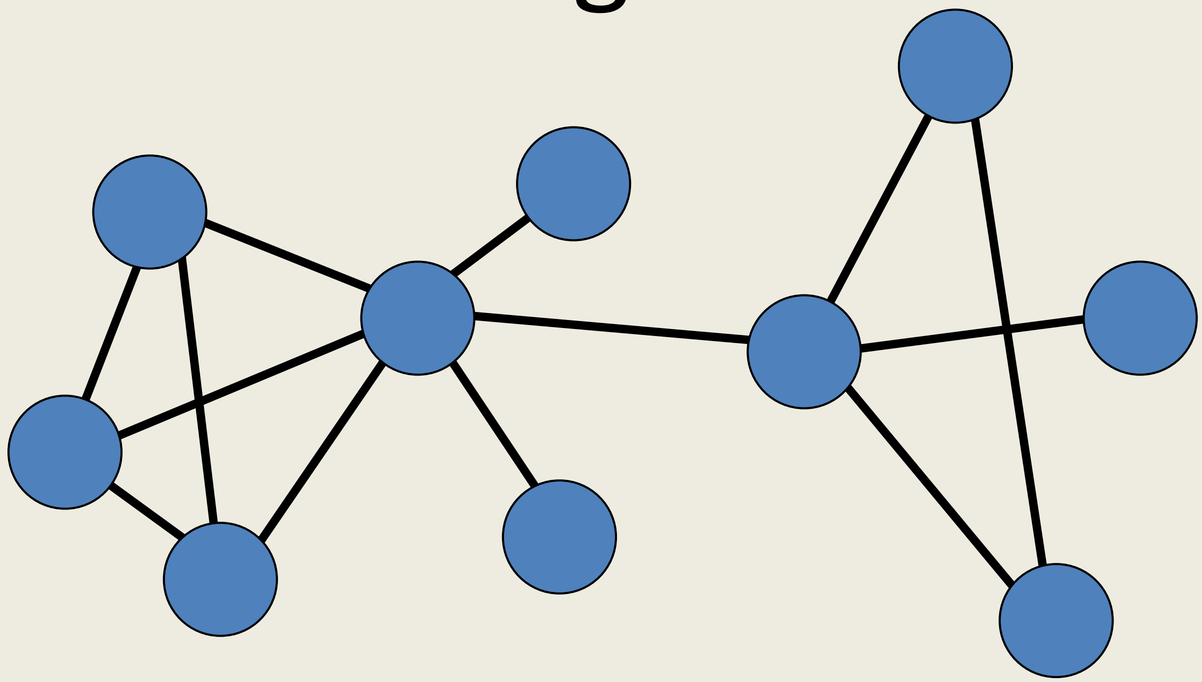
Find a minimum size set of vertices adjacent to all other vertices

Dominating set



NP complete problem so can't expect a P-time greedy algorithm to be optimal

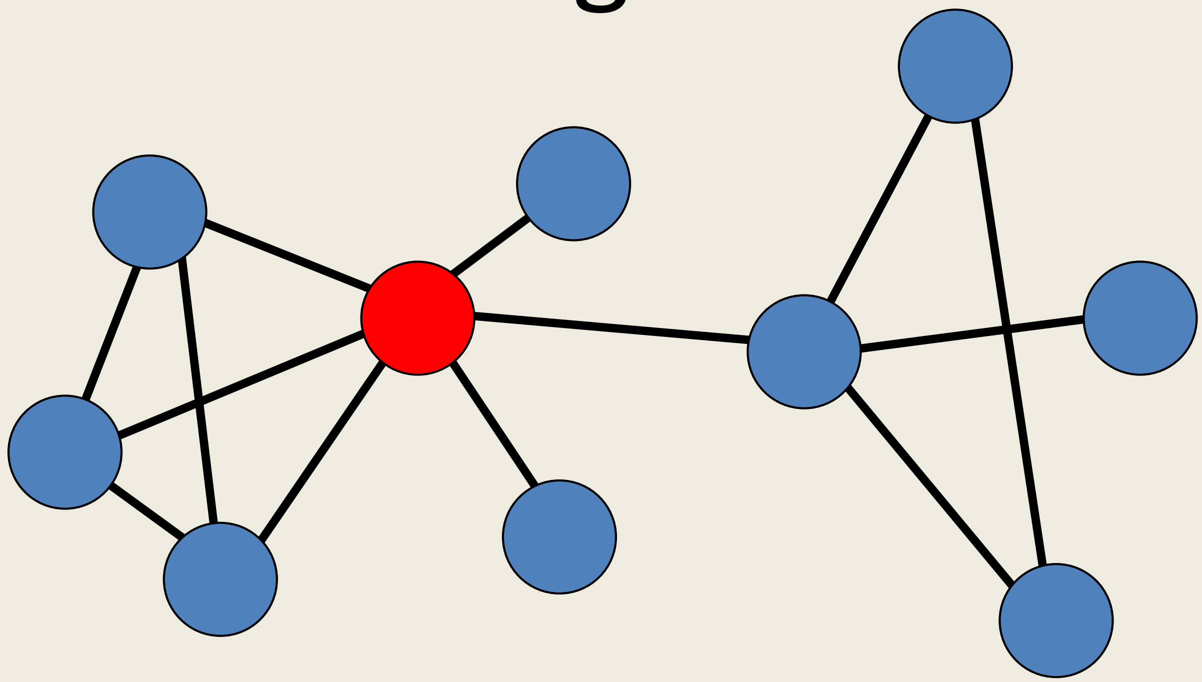
Dominating set



what can we do using a greedy rule?

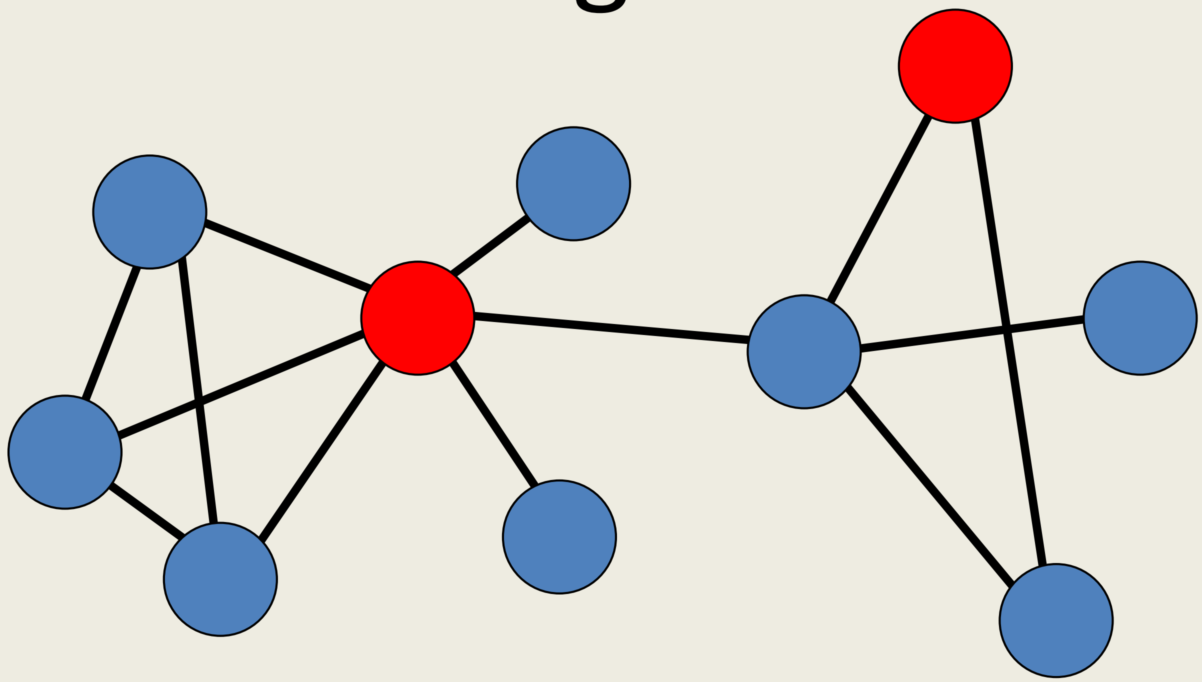
choose highest degree un-dominated vertex?

Dominating set



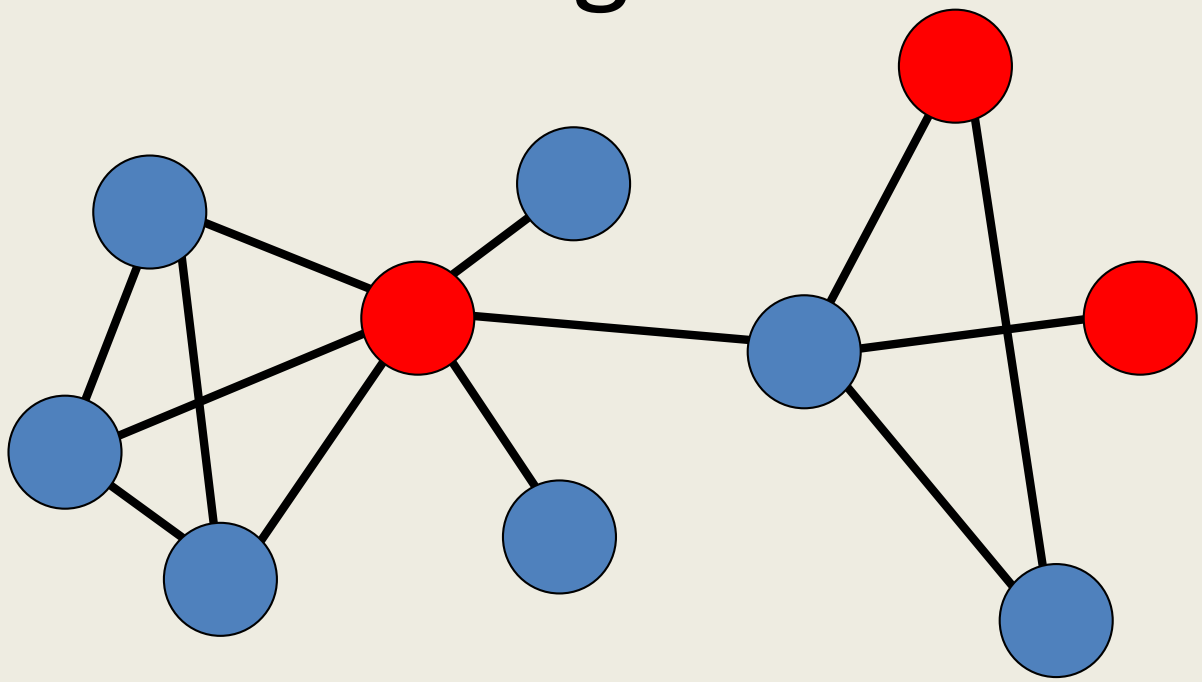
choose highest degree un-dominated vertex?

Dominating set



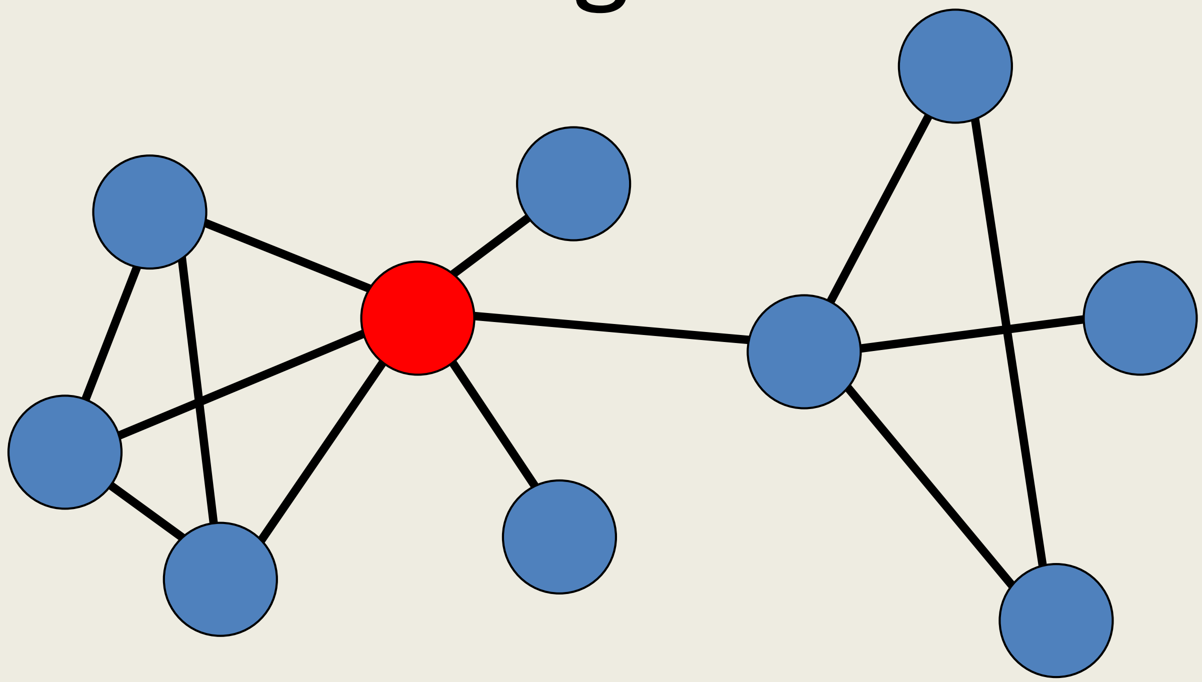
choose highest degree un-dominated vertex?

Dominating set



choose highest degree un-dominated vertex?

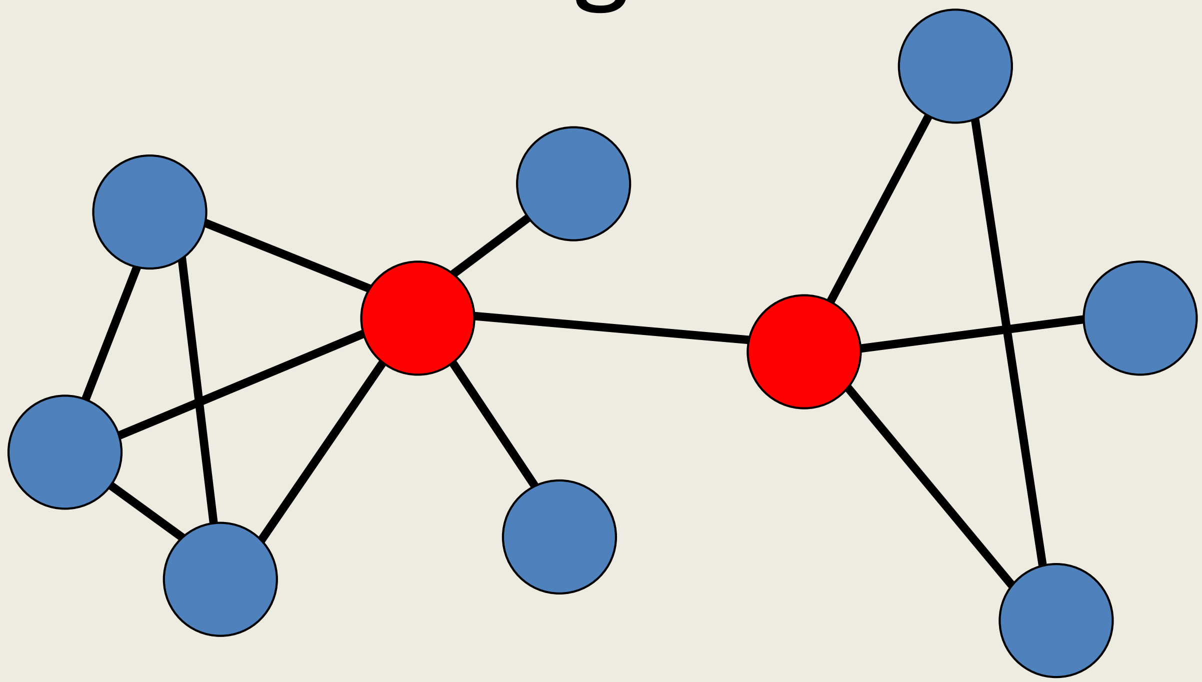
Dominating set



choose vertex with maximum marginal gain?

(gives largest increase in size of dominated set)

Dominating set

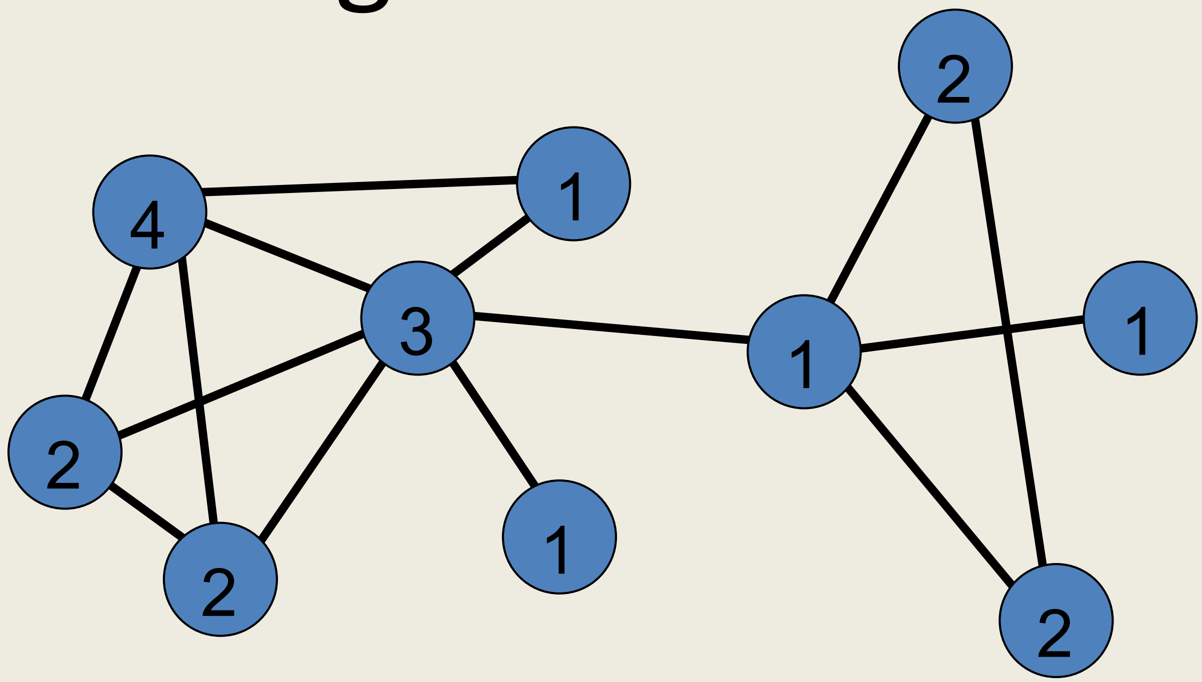


choose vertex with maximum marginal gain?

(gives largest increase in size of dominated set)

works very well in practice

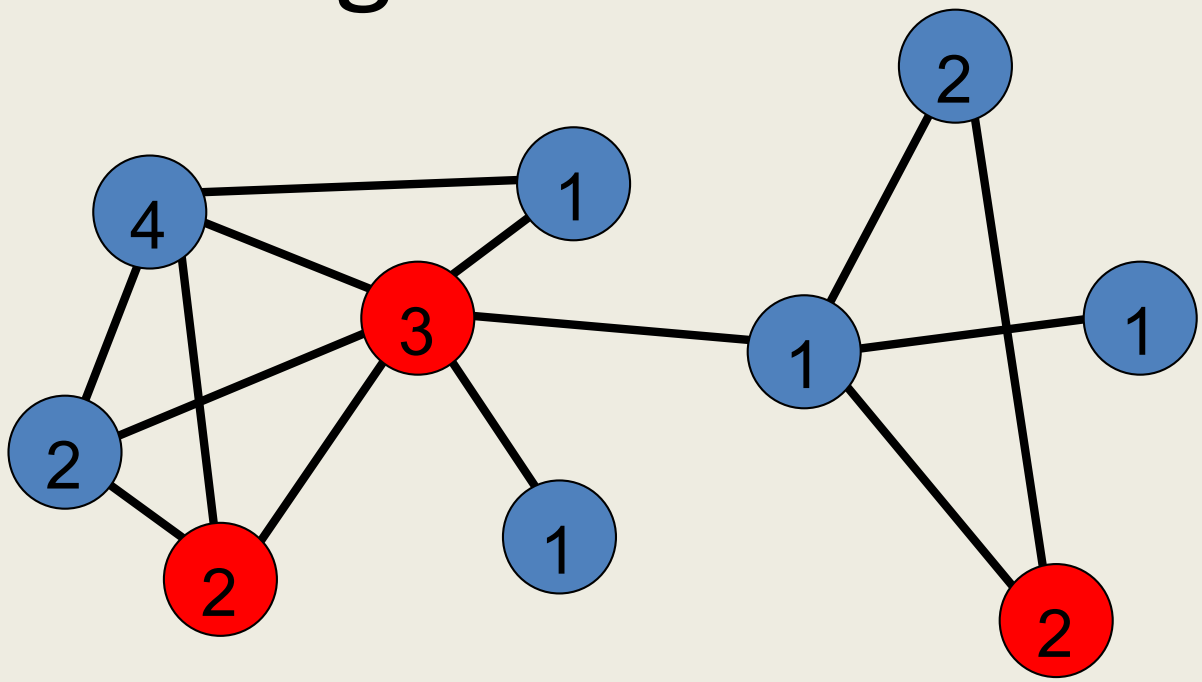
Target set



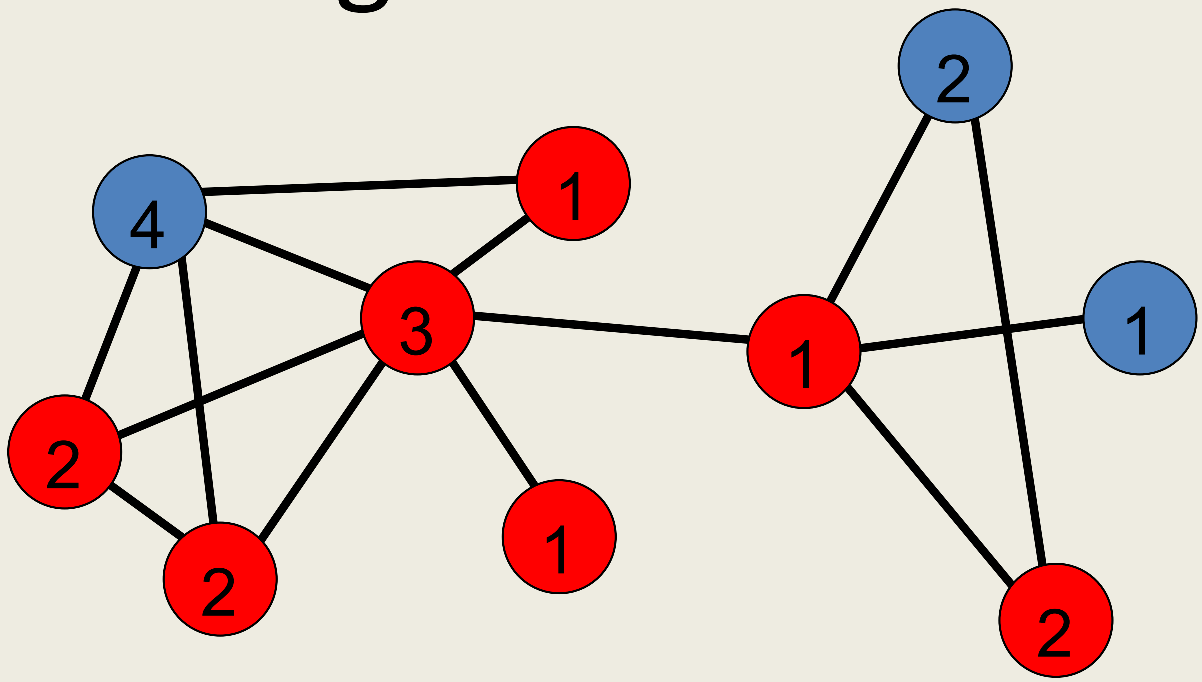
each vertex has an activation threshold (number of active neighbours needed to activate it)

find minimum size set of vertices that will activate all other vertices (another NP-complete problem)

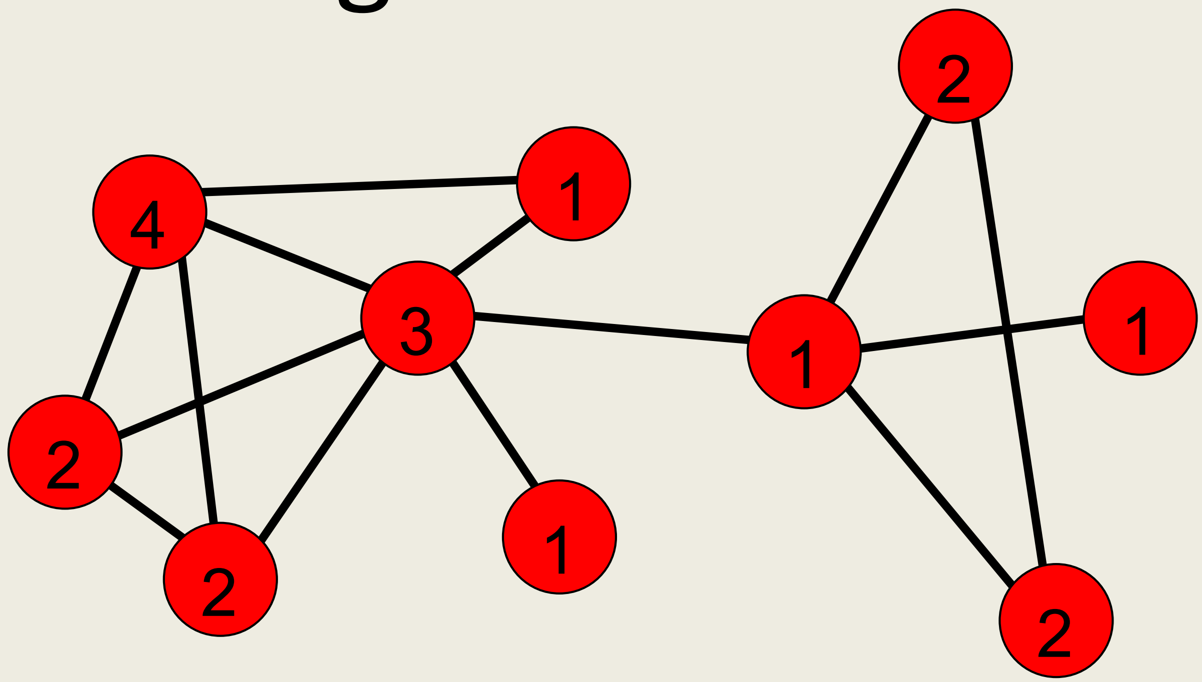
Target set



Target set



Target set



Target set

greedy rule:

for each vertex compute d(v) = degree – threshold

if threshold is greater than degree, a vertex can’t be activated by neighbours, so must be in target set

remove vertex v with smallest non-negative d(v)

degrees of remaining vertices decrease until all are < 0

put these remaining vertices into target set

Target set

correctness: does the greedy target set algorithm given here really produce a target set? Answer is YES

optimality: can't expect optimality using a P-time greedy algorithm but we can prove an upper bound on the size of the target set produced

(k is maximum threshold)

$$\Sigma_{v \text{ in } V} (k / \text{degree}(v) + 1)$$