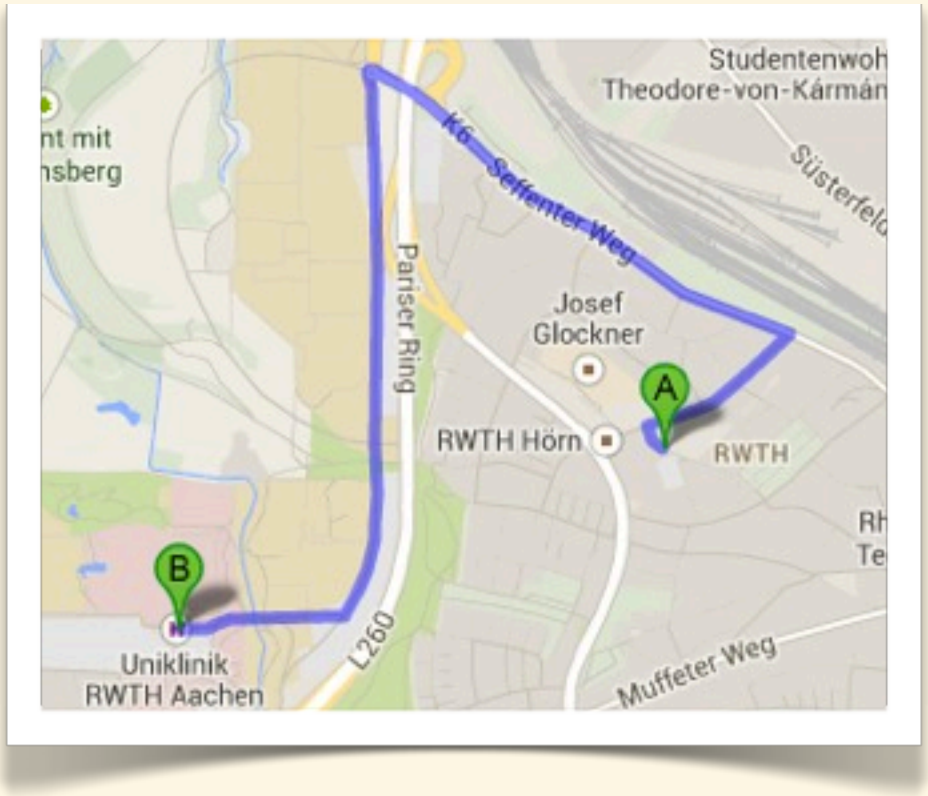
Shortest Paths In Undirected Planar Graphs With Nonnegative Weights

by Sascha Vincent Kurowski

The Problem

- Shortest paths from a given source node to all other nodes in the given graph
- For simplicity: **nonnegative** weights
- Applications:
  - Navigation between physical locations
  - Reaching goal state in state set (AI)
  - Minimize delay in a network
  - Plant and facility layout



Priority Queue

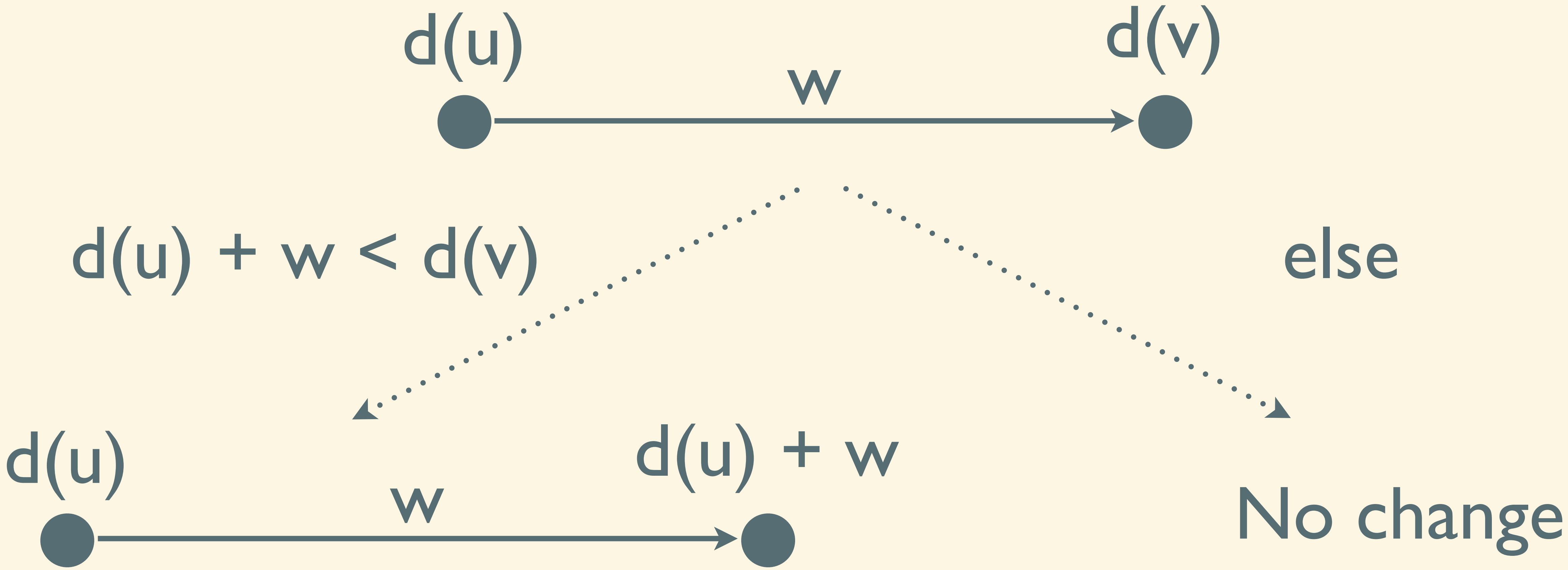
- Regular queue including **priorities** associated with each element
- **Fast implementation** using Fibonacci-Heap:

updateKey(Q, x, k)	updates key of x to k	O(1)
minItem(Q)	returns item with minimum key	O(log n)
minKey(Q)	returns key of minItem(Q)	O(log n)

Dijkstra's Algorithm

- Mark all nodes as unvisited
- Label the source node with 0, all others with ∞
- Repeat |V| times:
  - Choose the unvisited node v with **minimal label**
  - **Relax** all outgoing edges
  - Mark v as visited

Relaxing an edge



Dijkstra's Algorithm: Running Time

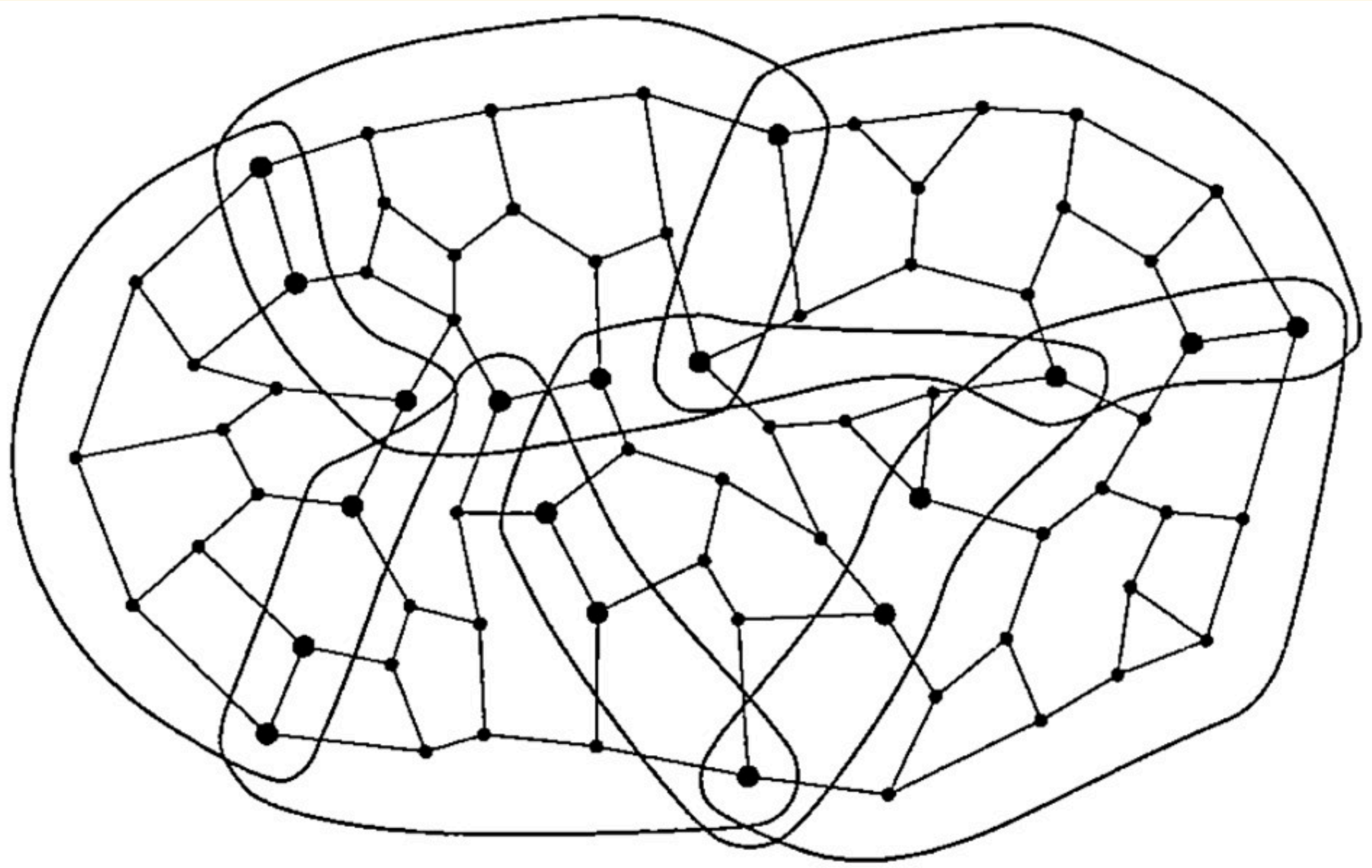
- Initialization in O(|V|)
- Repeat O(|V|) times
  - Choosing the node with smallest label: **O(log |V|)** using Fibonacci-Heap
  - Relaxing the edges: In total O(|E|) because every edge is relaxed only once
- Total time: O(|E| + |V| log |V|)

for planar graphs:
O(|V|)

- Fastest algorithm for any graph with nonnegative weights

Division of planar graphs

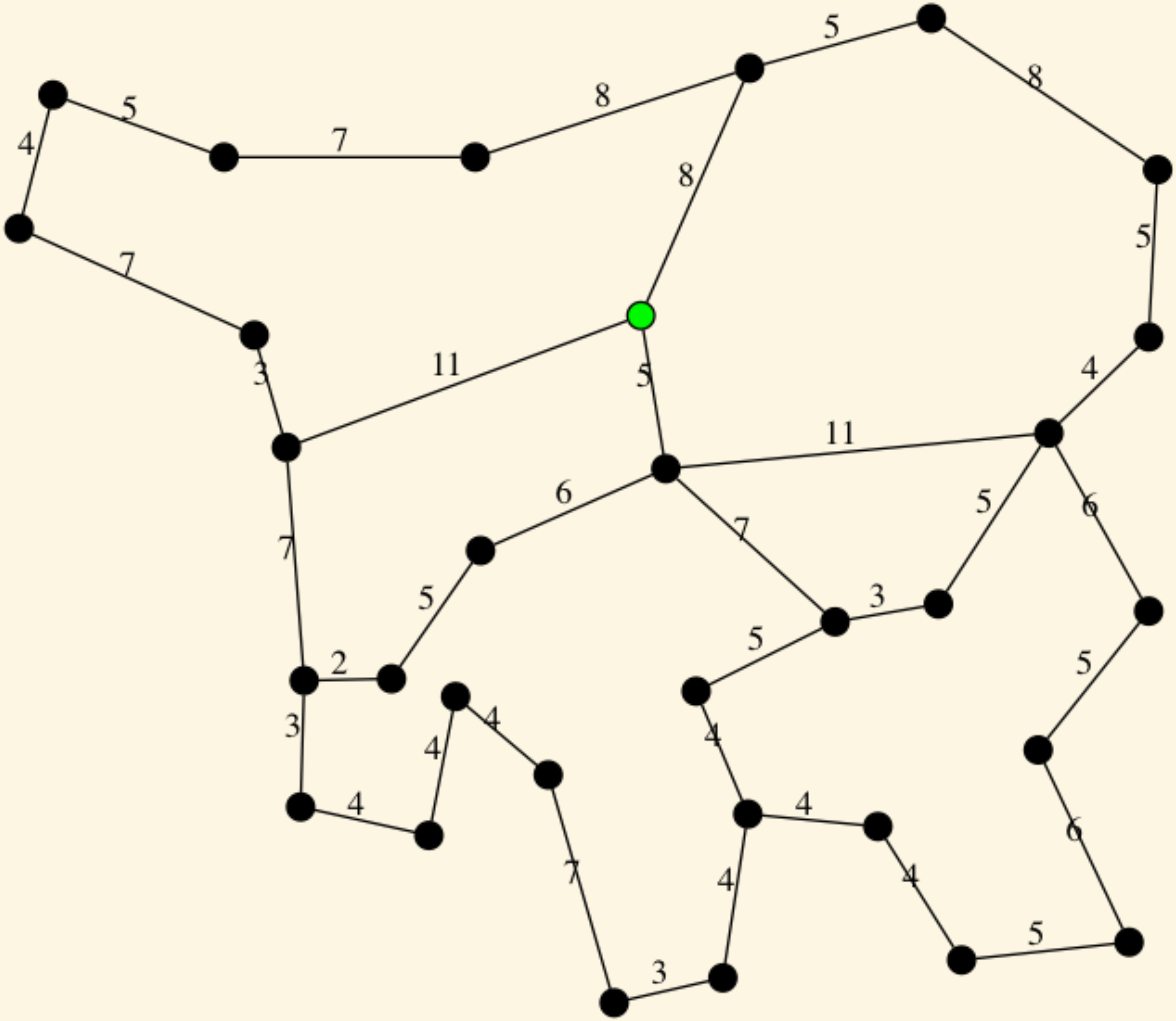
- **Partition of edge-set** into two or more subsets, called regions
- Node is contained in a region if some edge of the region is incident to the node
- Nodes contained in more than one region are called **boundary nodes**
- r-Division of a planar graph
  - Division into $O(n/r)$ regions
  - Each region contains at most r nodes including at most $O(\sqrt{r})$ boundary nodes



Simplified Algorithm

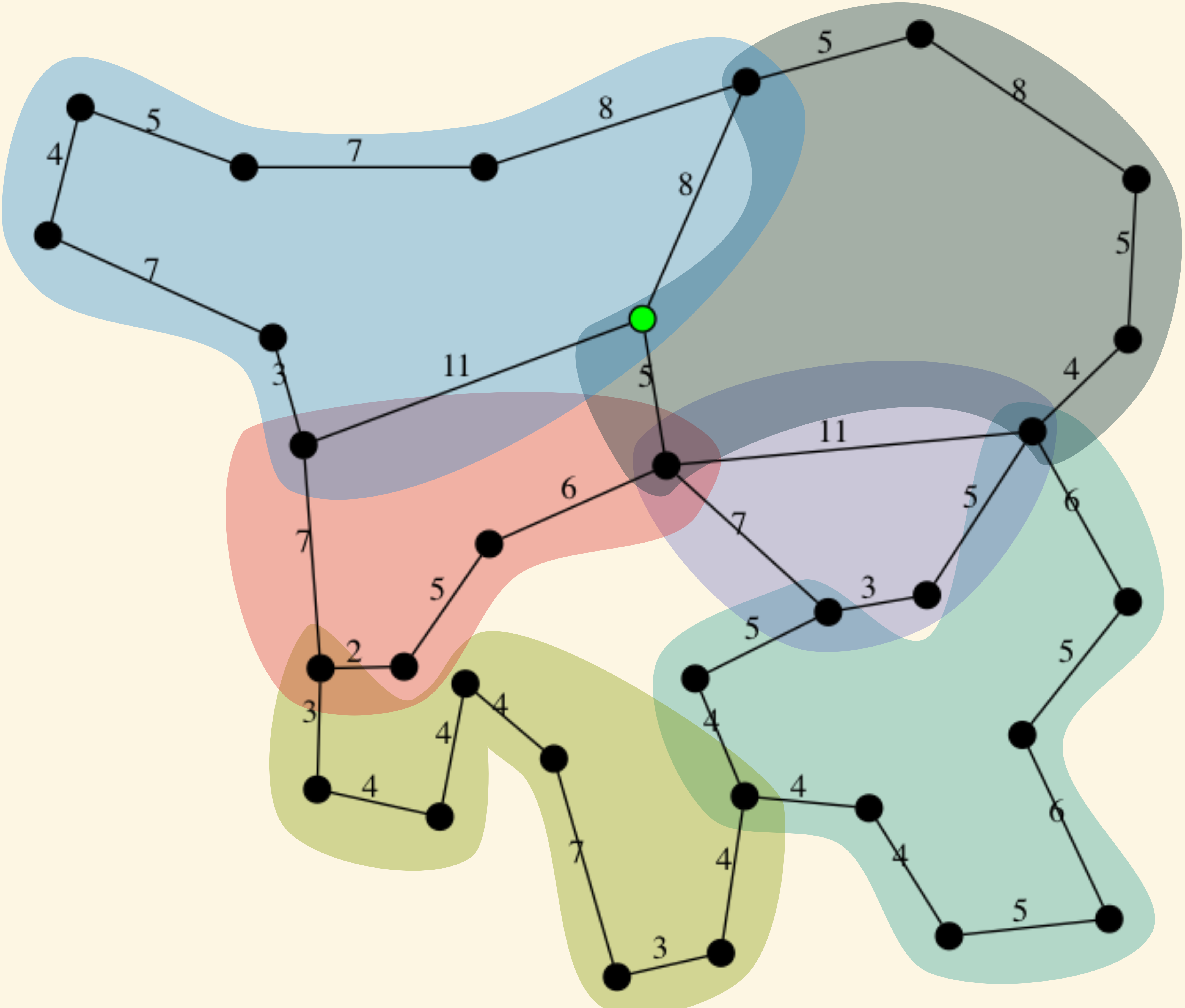
- Requires r-division with $r = \log^4(n)$
- Maintains **label for each node** (like Dijkstra)
- Maintains **status for each edge** (activated / deactivated)
- Runs in **O(n log log n)**

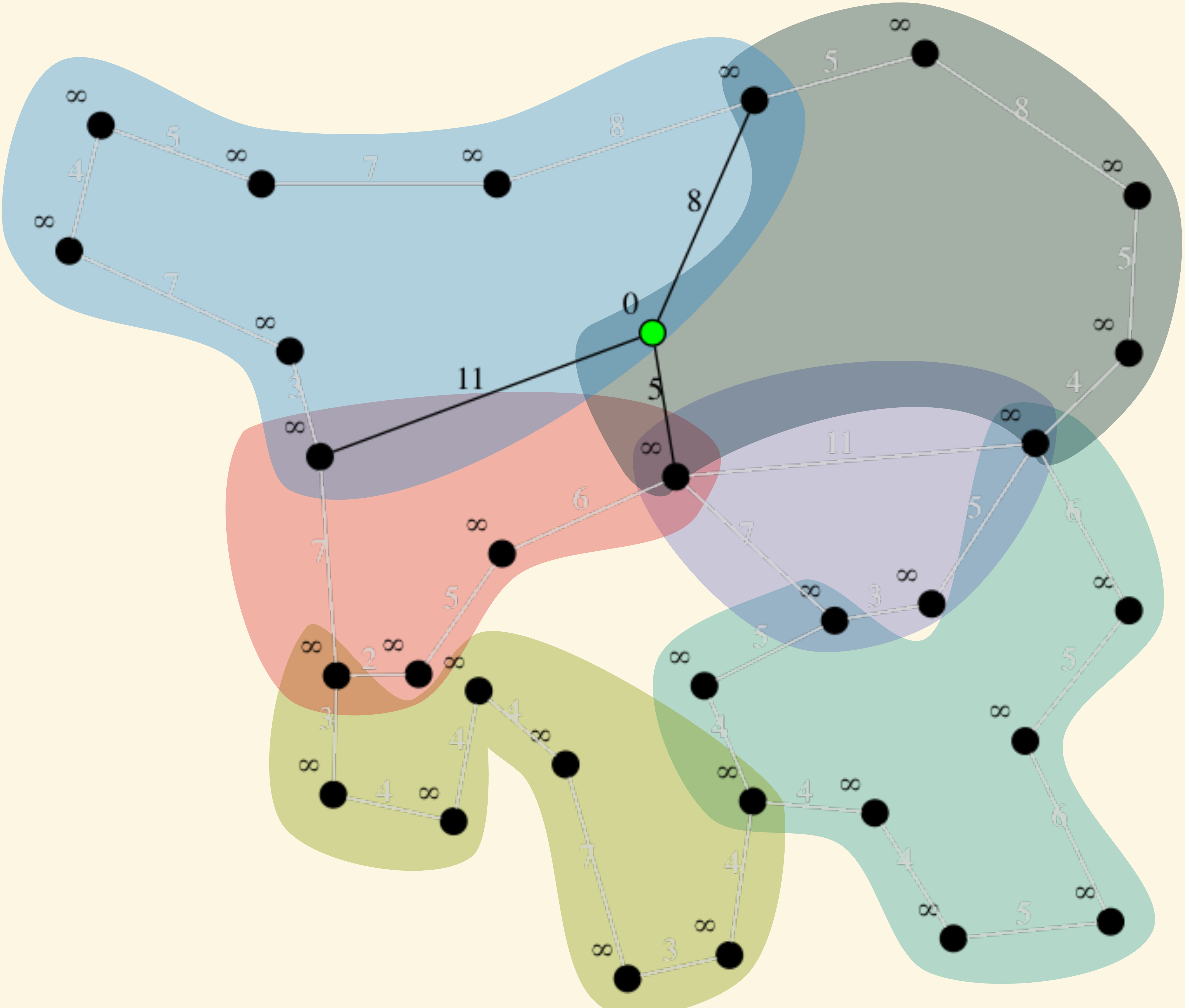

Saarland


5
8
5
8
4
7
8
5
7
11
5
4
3
11
6
7
5
6
7
3
5
5
2
5
4
3
4
4
4
4
6
4
7
4
5
3

Initialization

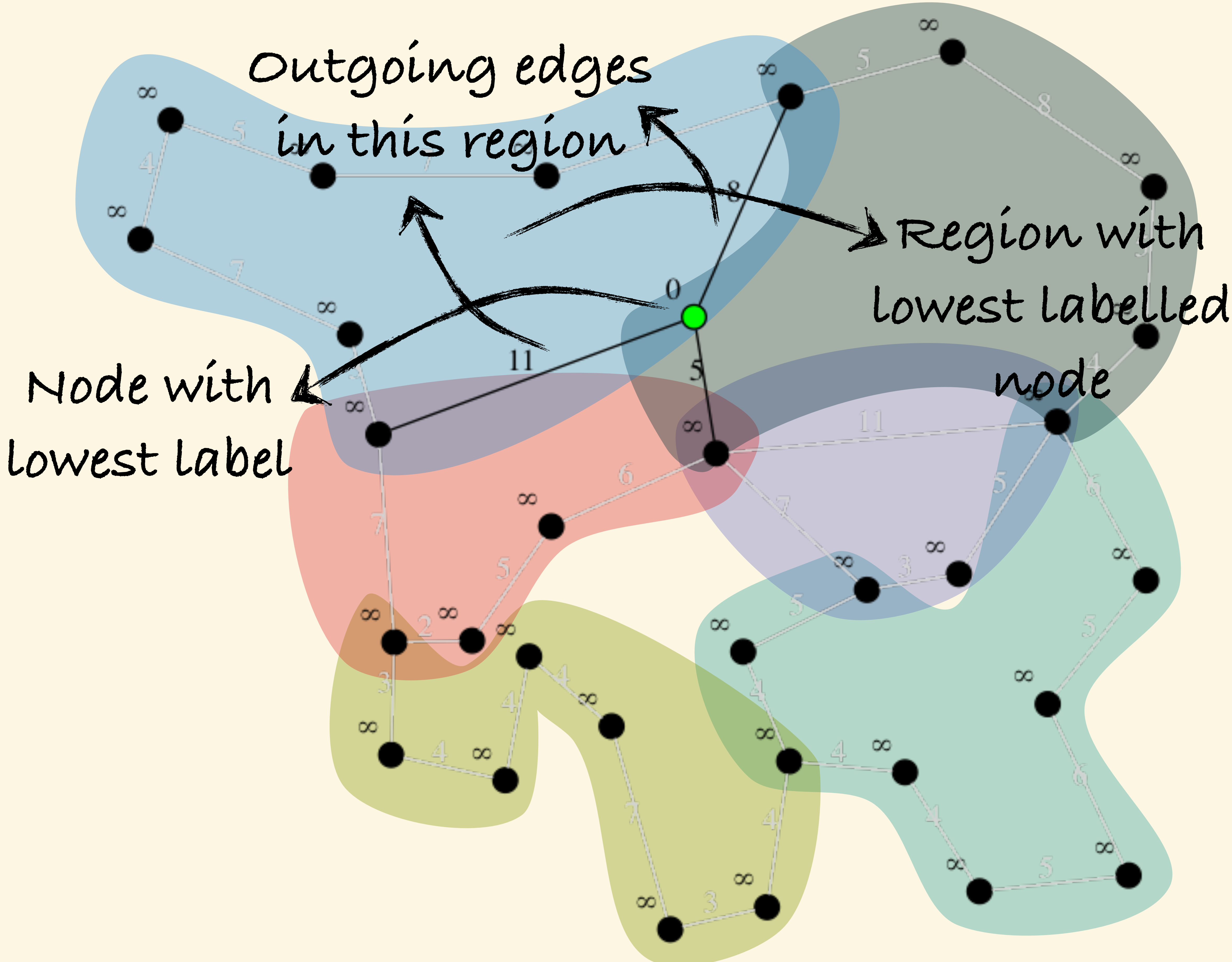
- Calculate needed r-division
- Deactivate all edges
- Set all node labels d(v) to ∞
- For source s
  - Set d(s) to 0
  - Activate all outgoing edges

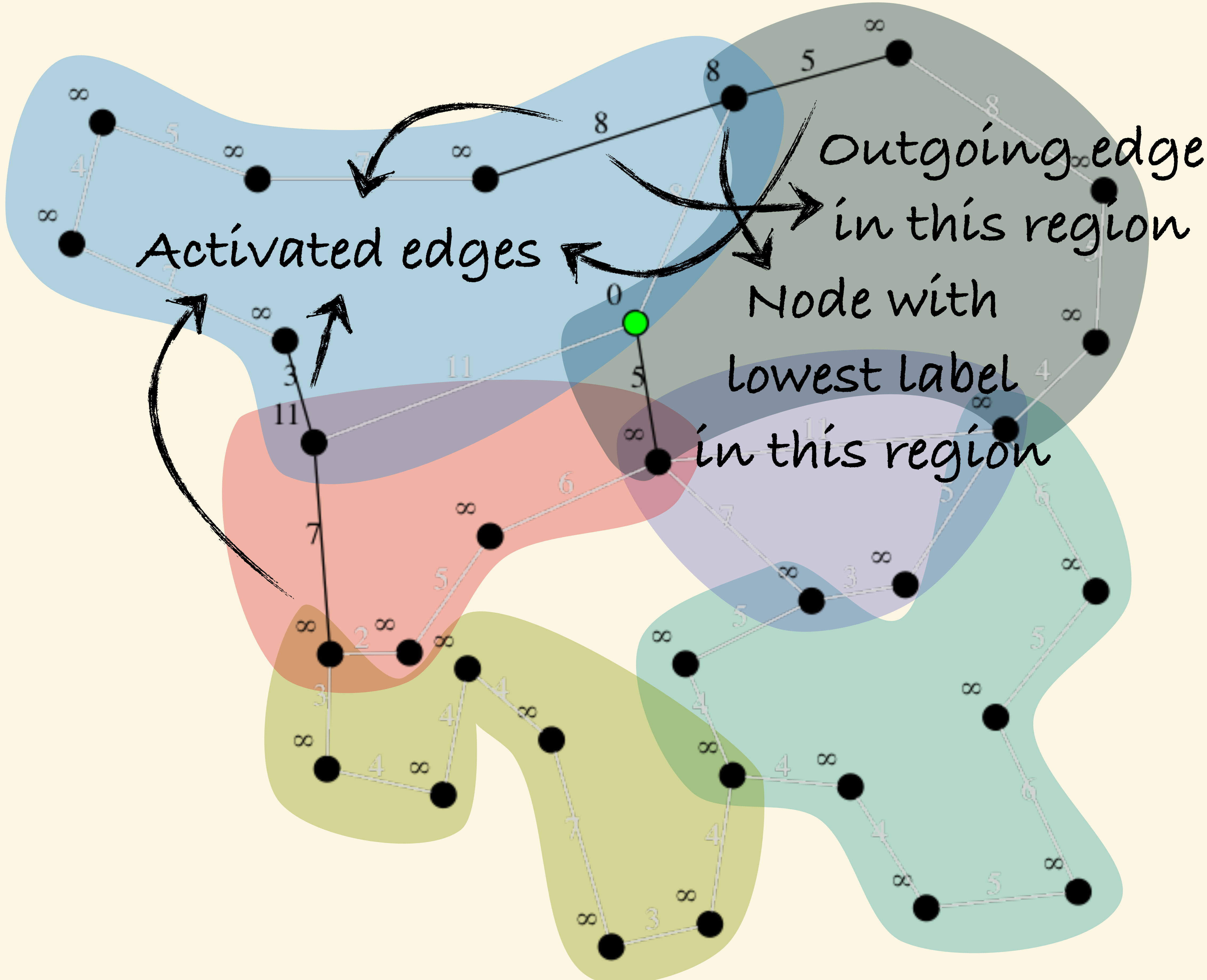

5
8
8
5
7
4
8
5
7
11
4
3
5
11
6
5
6
7
7
5
3
5
2
5
4
3
4
4
4
4
6
7
4
4
5
3

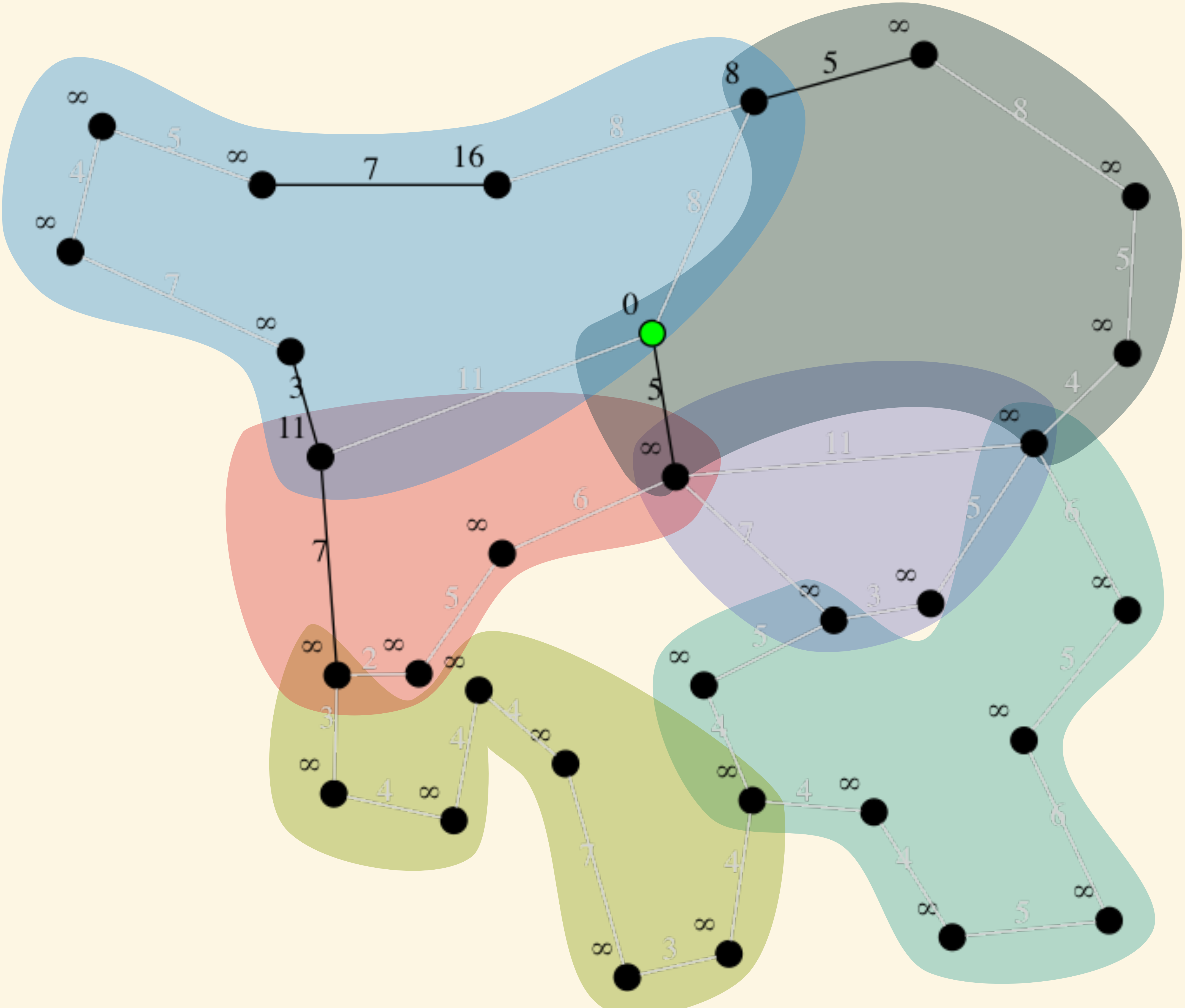

0
8
11
5

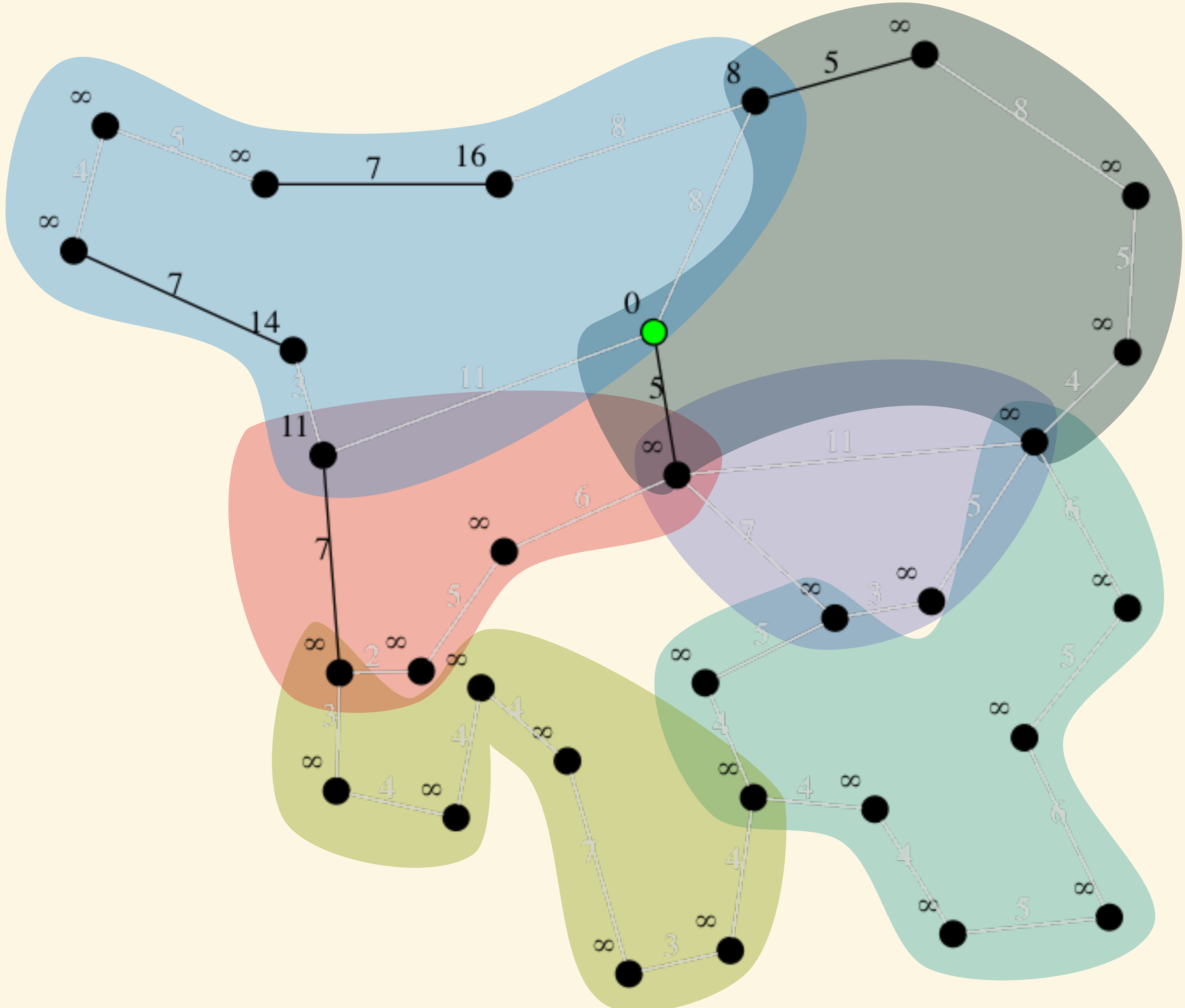
Algorithm

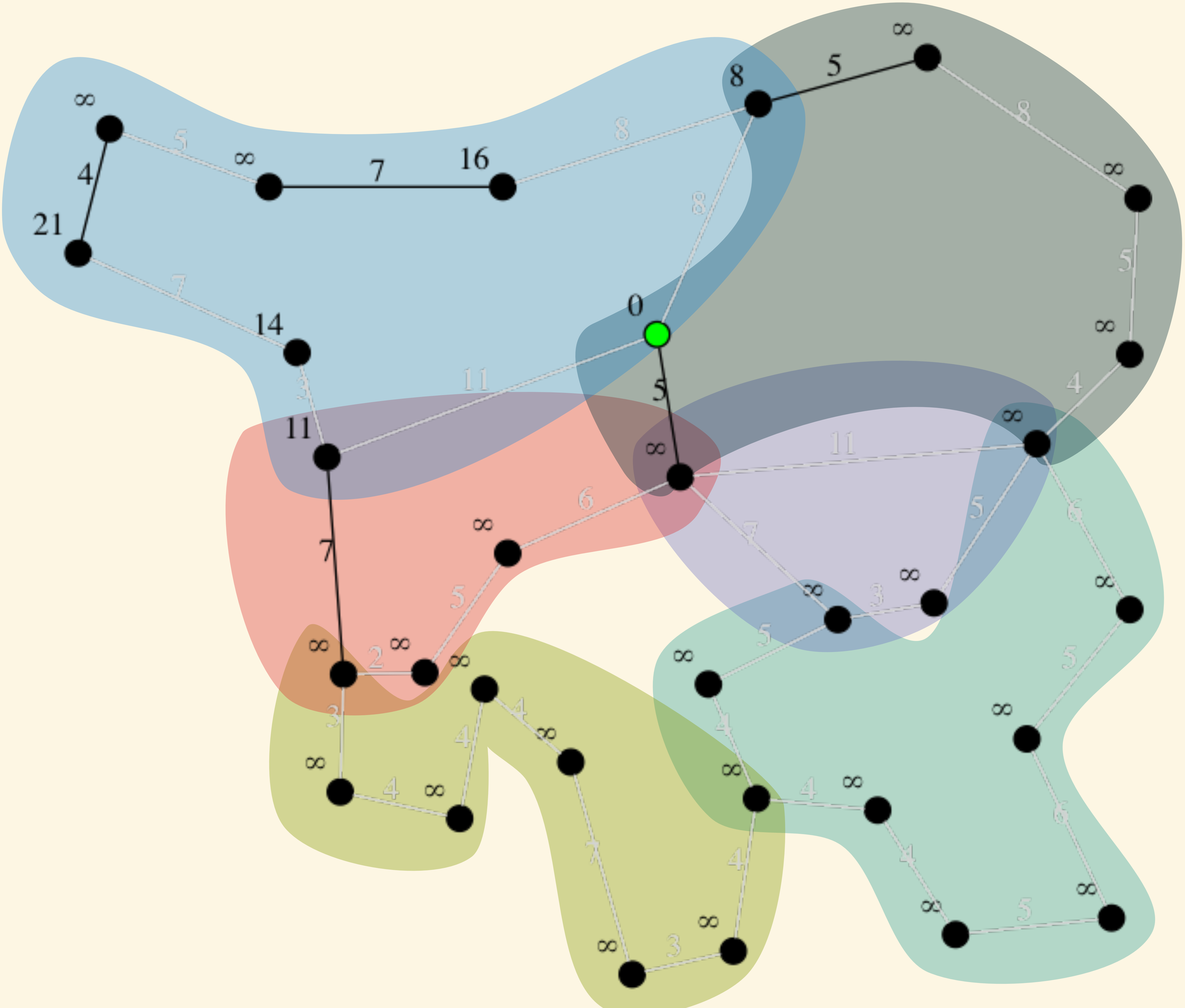
- Repeat:
  - Step 1: Select the region containing the lowest-labeled node that has active outgoing edges in the region
  - Step 2: Repeat log n times (if possible):
    - Step 2a: Select lowest-labeled node v in the current region with outgoing edges in the region
    - Step 2b: Relax and deactivate all its outgoing edges vw in that region
    - Step 2c: Foreach of the endpoints w: If relaxing the edge vw

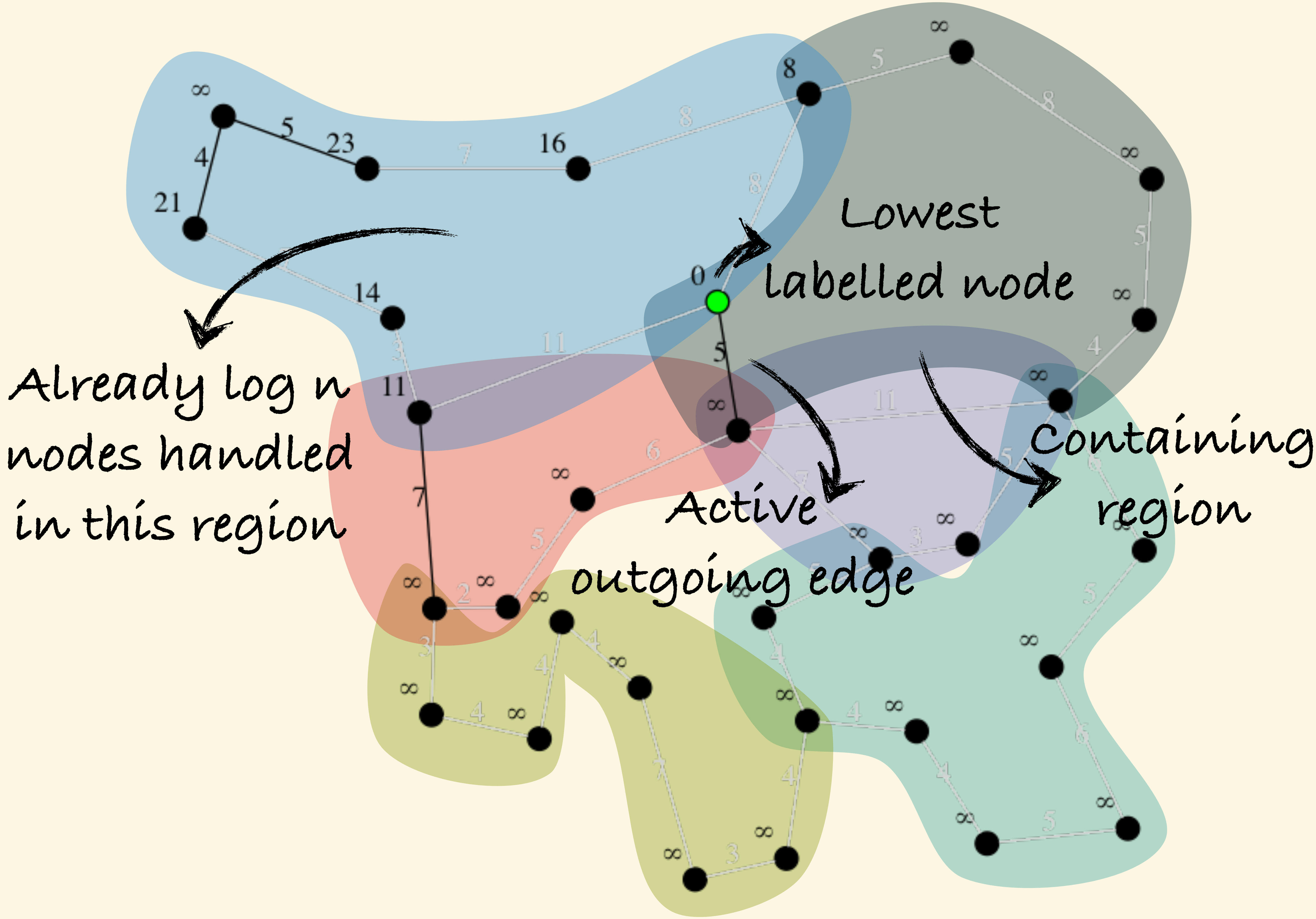

Outgoing edges
in this region
Region with
lowest labelled
node
Node with
lowest label

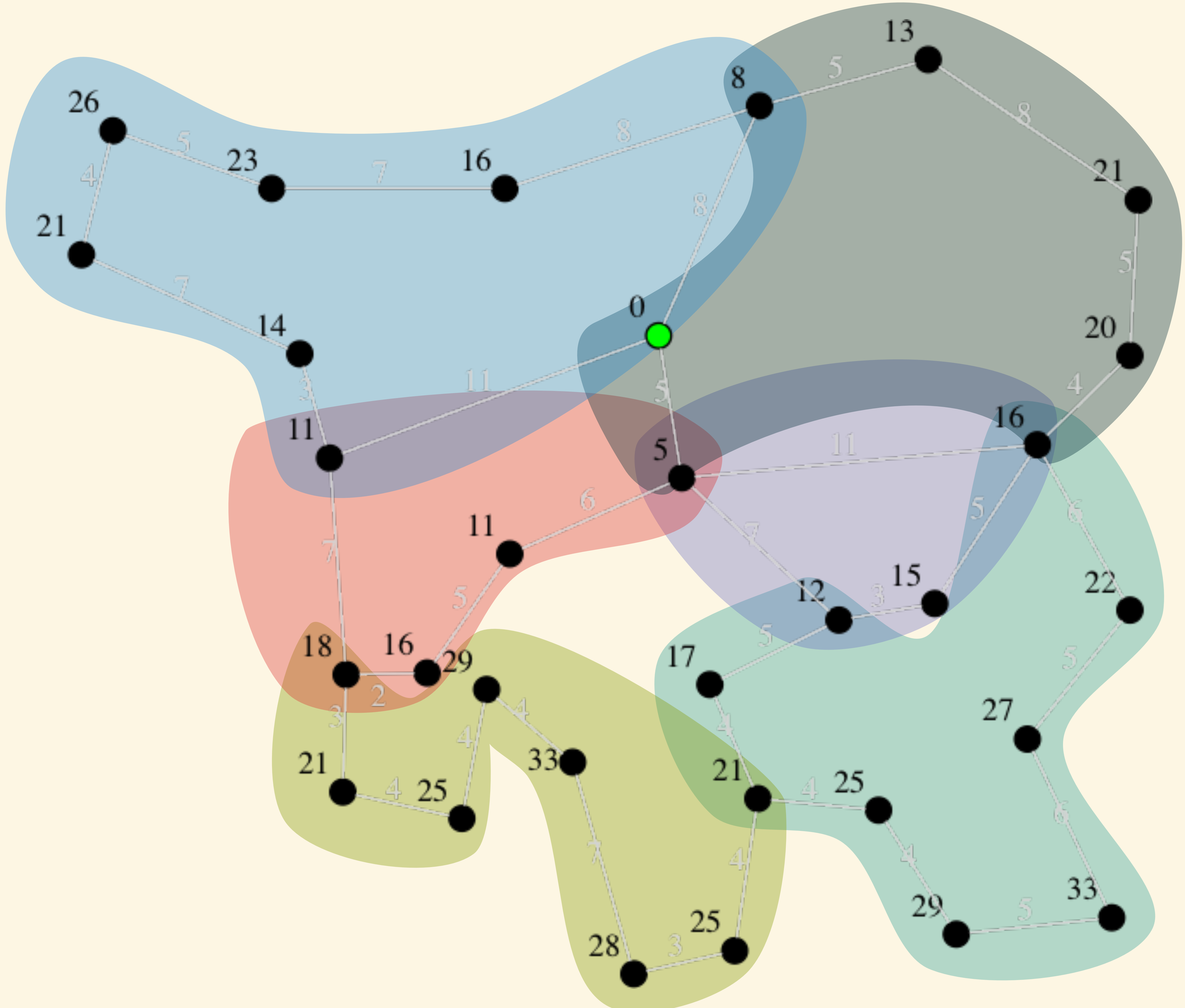

Outgoing edge in this region
Activated edges
Node with lowest label in this region


8
5
∞
16
7
0
5
3
11
7


∞
8
5
∞
16
7
∞
∞
14
7
0
5
11
11
7
∞


∞
8
5
∞
16
7
4
21
0
14
5
11
7
∞
2
∞


Lowest
labelled node
Already log n
nodes handled
in this region
Active
outgoing edge
Containing
region



13
8
26
23
16
21
21
0
20
14
11
16
5
11
22
15
12
18
16
29
17
27
21
33
21
25
25
33
29
25
28

Correctness

- Shortest path conditions:

1. $d(s) = 0$
2. every label $d(v)$ is an **upper bound** on the distance
3. every edge is relaxed

Shortest Path Condition 1
d(s) = 0

- At initialization d(s) is set to 0
- Every edge has a nonnegative weight
- Nodes' labels are only updated when relaxing an edge to them
- Therefore d(s) never changes

Shortest Path Condition 2

every label d(v) is an upper bound on the distance

- Initially every label (except for d(s)) is ∞
- The labels only change in step 2b
- Assuming inductively d(u) and d(v) are upper bounds on distance to u and v, new value d'(v) is also an upper bound
- Full proof by induction on number of steps of algorithm that have been executed

Shortest Path Condition 3

every edge is relaxed

- Proof that if an edge is inactive, it is relaxed
- Holds after initialization
- Algorithm deactivates an edge right after relaxing it
- Existing inactive edge vw might become unrelaxed when the labels of its endpoints change
- This may happen when relaxing an edge leading to v
- In the same step the algorithm activates vw

Shortest Path Condition 3

every edge is relaxed

- Proof that after termination, all edges are deactivated
- Obviously true, because the algorithm stops only when it can't select a new region with active outgoing edges anymore

Recursive r-division

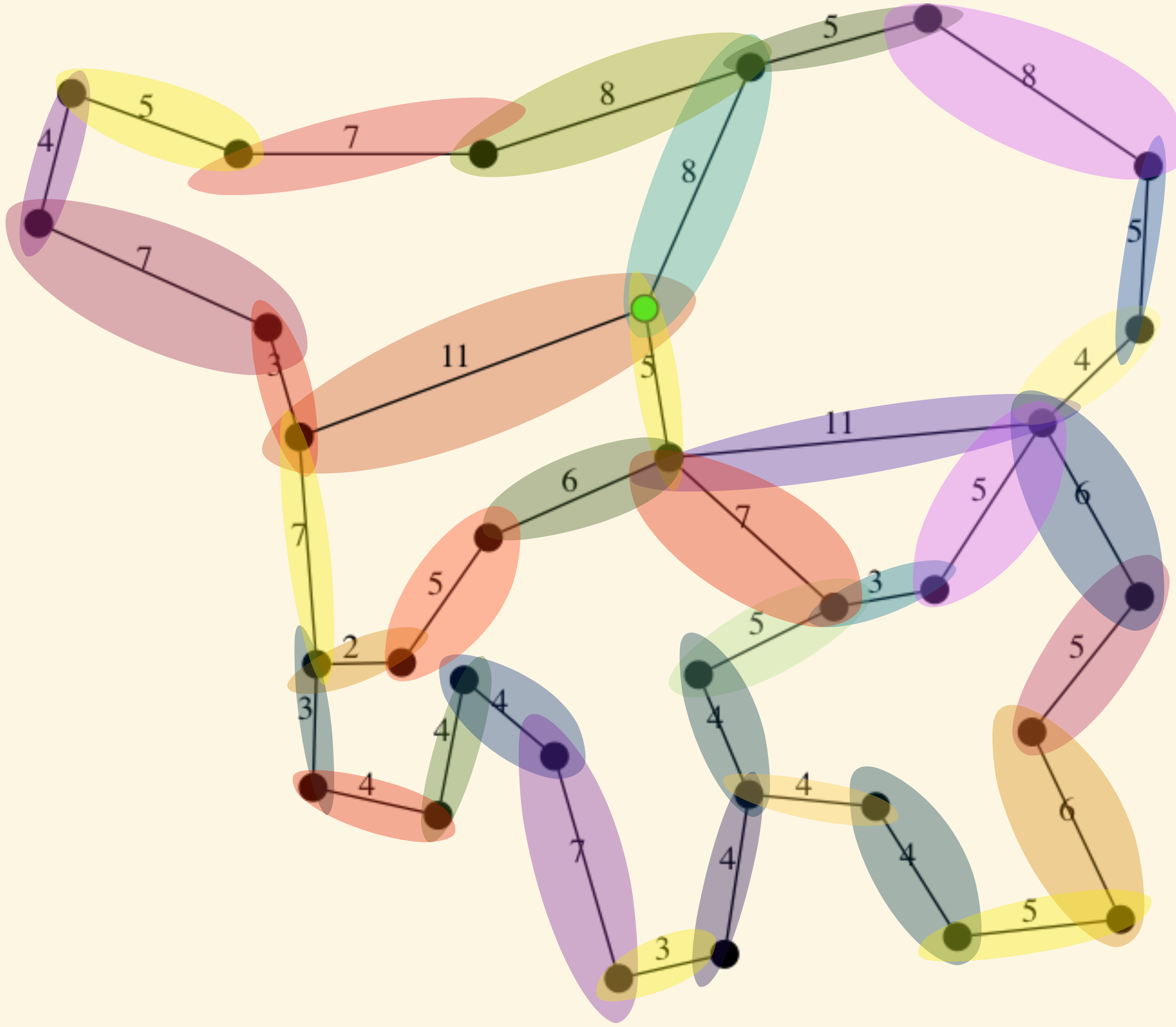
- (r, s)-division of an n-node graph:
  - division into O(n/r) regions, each containing r^(O(1)) nodes, each having at most s boundary nodes
- Recursive r-division of an n-node graph G:
  - Repeatedly divide the regions of an (r, s)-division into smaller and smaller regions
  - Contains one region consisting of all of G

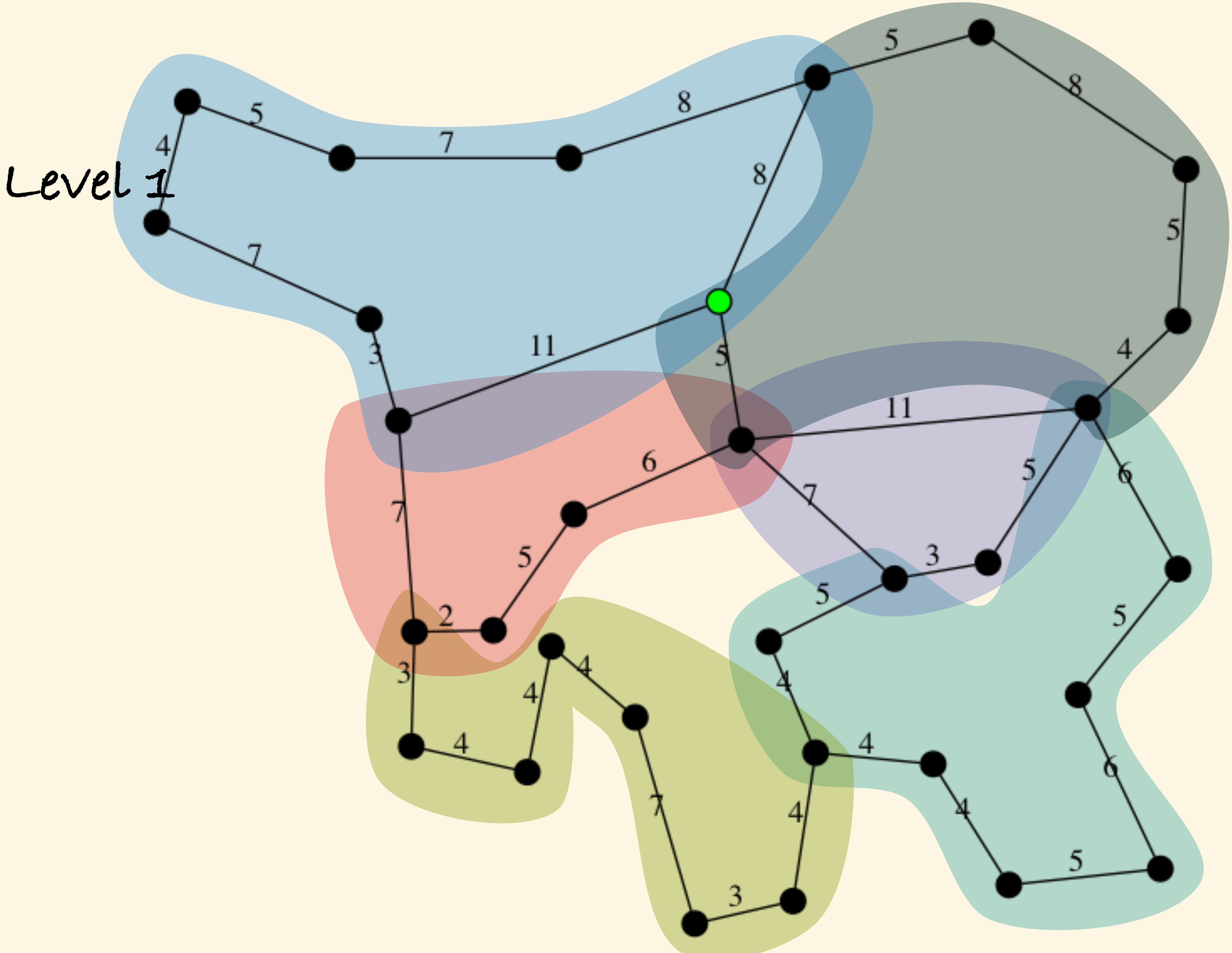
Notations

- For two regions R1 and R2 of different divisions, R1 is an ancestor of R2 if R1 contains R2
- Immediate ancestor is called the parent
- Descendants and children defined analogously
- Region without children: Atomic Region
  - For this algorithm atomic regions consist of exactly one edge, denoted R(uv)
- Level of atomic region is 0, for nonatomic regions maximum of children's levels

Example

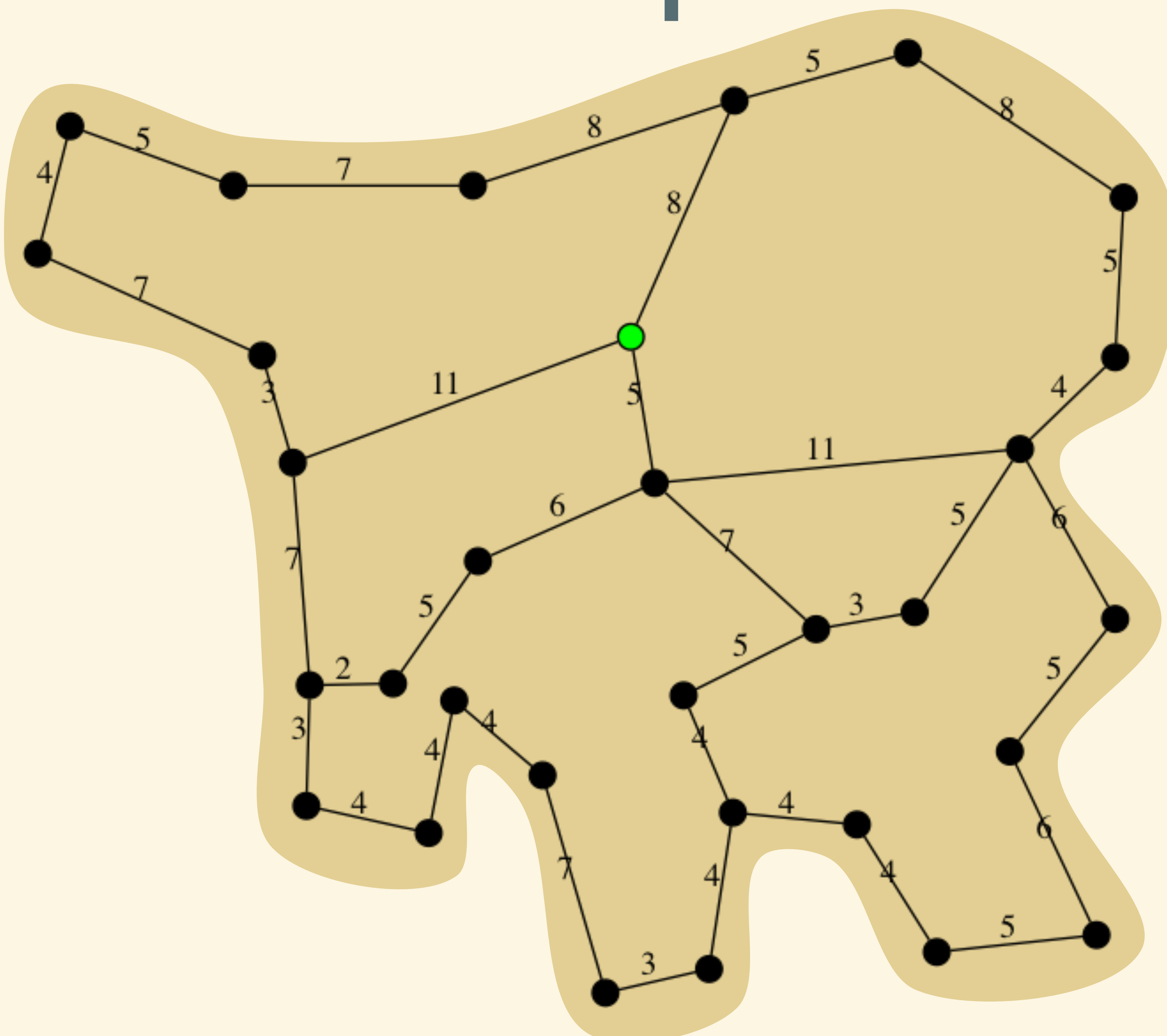
Level 0




Level 1
5
8
5
8
7
8
4
5
7
5
3
11
5
4
11
6
7
5
6
7
3
5
2
5
3
4
4
5
4
4
4
6
7
4
4
5
3

Example

Level 2



Formal Algorithm

- Maintains a priority queue Q(R) for each region R of the recursive division of G
- For nonatomic regions R, Q(R) contains all children of R
- For atomic Regions R', Q(R') contains the single edge uv contained in R'
  - Associated key is either
    - Label of tail of the edge
    - or ∞ to denote a deactivated edge

Formal Algorithm

- Goal:
  - Ensure that for any region R
  - minKey(Q(R)) is minimum distance label d(v) over all active edges vw in R
- Two procedures:
  - Process(Region R)
  - GlobalUpdate(Region r, Item x, Value K)

```
Process(R)
// R is a region
If R contains a single edge uv then // R is atomic
    if d(v) > d(u) + w(uv) then
      d(v) := d(u) + w(uv)
      foreach outgoing edge vw of v
        GlobalUpdate(R(vw), vw, d(v))
 Else // R is nonatomic
    Repeat α_i times or until minKey(Q(R)) is ∞
      R' := minItem(Q(R))
      Process(R')
      updateKey(Q(R), R', minKey(Q(R')))
```

```
GlobalUpdate(r, x, k)
// R is a region, x is an item of Q(R) and k is a value
    updateKey(Q(R), x, k)
    If the updateKey operation reduced minkey(Q(R))
    then
       GlobalUpdate(parent(R), R, k)
```

The algorithm

- Initialize all labels and keys to ∞
- Assign label $d(s) := 0$ and foreach outgoing edge sw call GlobalUpdate(R(sw), sw, 0)
- Until minKey(Q(R(G))) = ∞
  - Process(R(G))

Execution

- Progress(R(G))
  - Progress(R’)
    - Progress(sw)
    - Progress(sv)
    - ...
  - Progress(R”)
    - Progress(wu)
    - Progress(wt)
    - ...

Invocations of Progress

level i	calls α_i	time per invocatio
2	1	O(log n)
1	log n	O(log n log log n)
0	0	O(1)

Truncated invocation of Process

- Truncated invocation of Process(R) when
  - MinKey(R) = ∞ after invocation
- Every level 0 invocation is truncated
- Exactly one level 2 invocation is truncated (the last one)

Execution

- Progress(R(G))
  - Progress(R’)
    - Progress(sw)
    - Progress(sv)
    - ...
  - Progress(R”)
    - Progress(wu)
    - Progress(wt)
    - ...

- Goal: Count the truncated invocations
- Charge them to one of the $O(n/\sqrt{r})$ boundary nodes
- Blame a pair of a region and a boundary node (R, v)

Blamed pairs

- (R(G), s)
- (R', v)
- (R', v')
- (R(uv), u)
- (R(uw), u)

Charging Scheme Invariant

- Have a charging scheme s+
- for any pair (R, v)
  - there is an invocation b of Process(R) so that **all invocations** charging to (R, v) are descendants of B or B itself

Invocations of Progress

level i	total number of invocations	time per invocation
2	O(n/log n)	O(log n)
1	O(n/log n)	O(log n log log n)
0	O(n)	O(1)

Thank you
for your attention

Any questions left?