CS301 Software Development

Instructor: Peter Kemper

Boruvka's Algorithm

Graphs and Minimal Spanning Trees

- Graph G = (V,E)
  - V set of vertices, E set of edges
  - here: undirected, edges can carry a numerical weight or cost
- Graph concepts
  - Reachability: Path from node v to w
  - Minimal spanning tree:
    - Subset of edges such that all vertices are connected, i.e. for all nodes v and w there exists a path from v to w and vice versa
    - Minimal: sum of weights is minimal
    - Typical assumption: weights are unique

How to generate a Minimal Spanning Tree (MST)?

- Observation:
  - If you have n already connected components which are trees and m left over edges connecting these (i.e. for each edge their end nodes are in different components), then you need to select n-1 out of m edges.
  - You can work towards an MST by adding one minimum weight (cost) edge connecting 2 components that are separate so far. This reduces the problem to n-1 trees and at most m-1 left over edges.

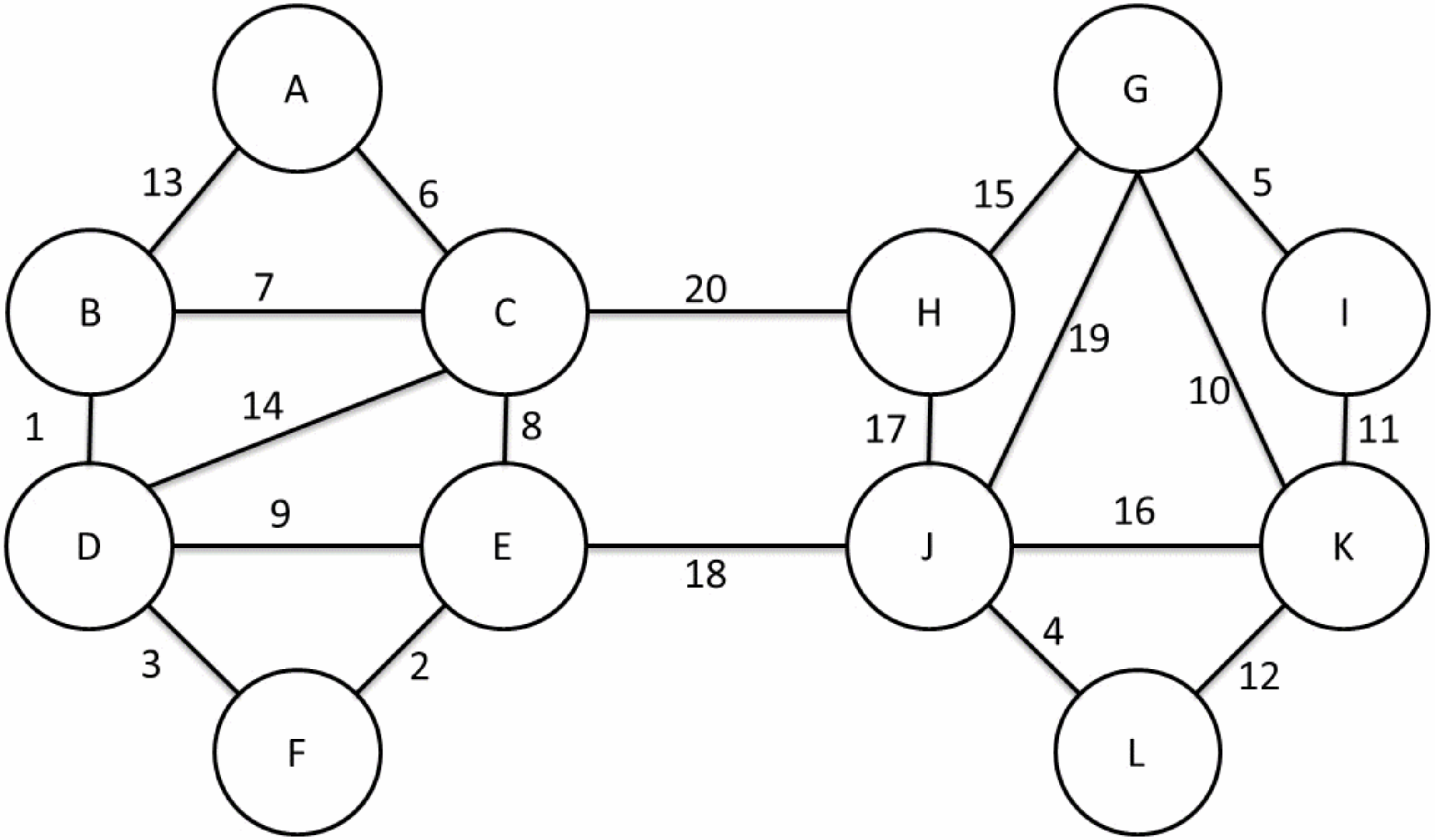
- For MST:
  - Select cheapest option at hand (locally), pick the cheapest edge
  - Never connect nodes that are in the same component
    - As they are already connected

How to generate a Minimal Spanning Tree (MST)?

- Variants of this idea
- Prim: “grow a single tree”
  - Select your favorite component and grow it by adding edges and thus nodes to it.
  - So: iterate over subset of edges expanding a single component.
- Kruskal: “grow a forest by growing one tree each step”
  - Select the cheapest left over edge and connect the components.
  - So: iterate over edges merging any 2 components
- Boruvka: “grow a forest by growing each tree each step”
  - Select the cheapest edge for each component and connect the components.
  - So: iterate over components and for each merge it with one adjacent component by selecting the cheapest edge

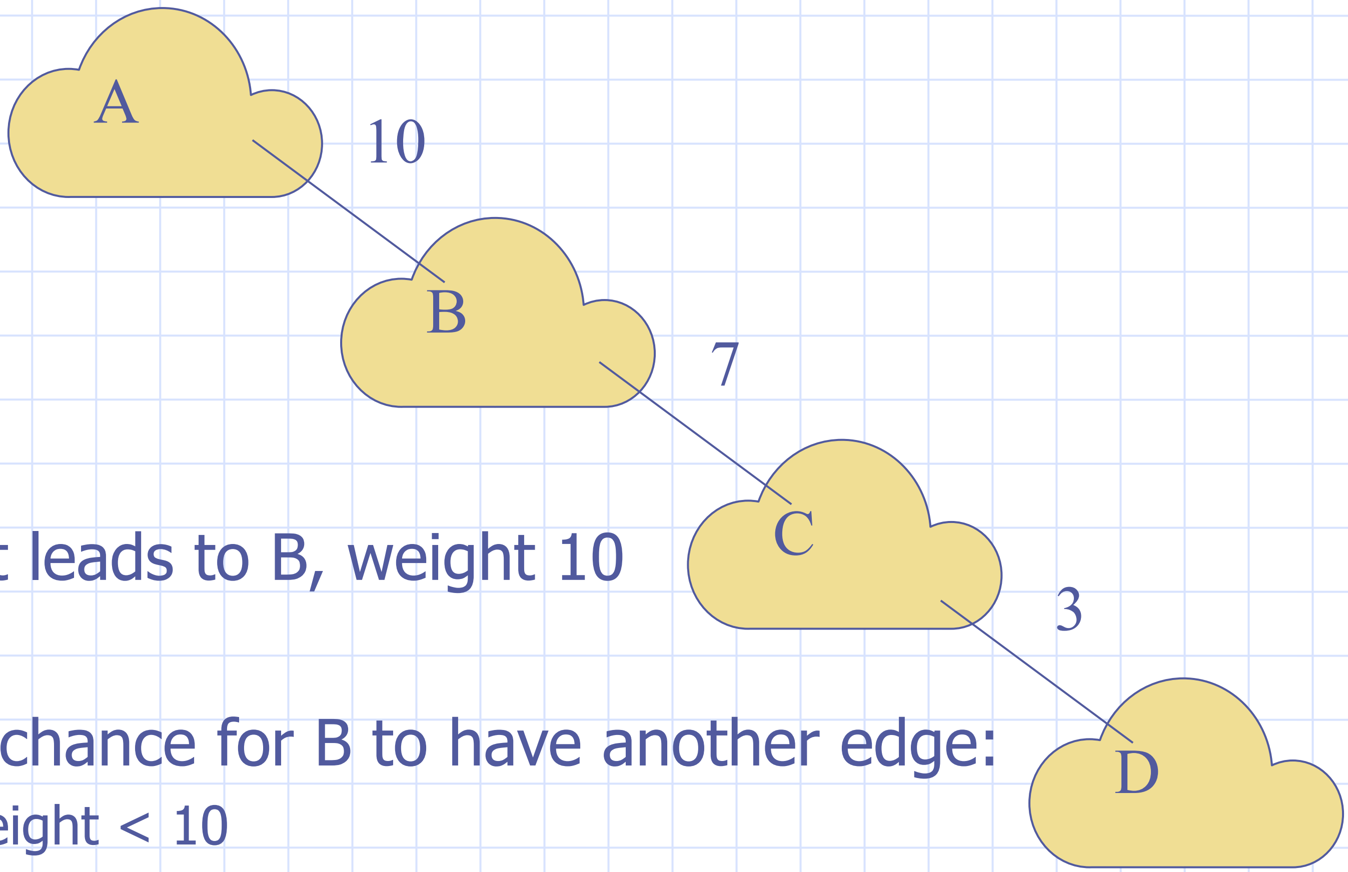
Algorithm and Animation from Wikipedia

- Credits: Alieseraj, cc, from Wikipedia



How do unique edge weights prevent cycles?

- Pick the cheapest edge leaving the component



- Say it leads to B, weight 10
- Only chance for B to have another edge:
  - Weight < 10
  - Since weights are unique, there are no equal weights, so there is no confusion on which one is the cheapest
- Edges are undirected, so could at most go back one.

```
algorithm Borůvka is
    input: A weighted undirected graph G = (V, E).
    output: F, a minimum spanning forest of G.

    Initialize a forest F to (V, E') where E' = {}.

    completed := false
    while not completed do
        Find the connected components of F and assign to each vertex its component
        Initialize the cheapest edge for each component to "None"
        for each edge uv in E, where u and v are in different components of F:
            let wx be the cheapest edge for the component of u
            if is-preferred-over(uv, wx) then
                Set uv as the cheapest edge for the component of u
            let yz be the cheapest edge for the component of v
            if is-preferred-over(uv, yz) then
                Set uv as the cheapest edge for the component of v
        if all components have cheapest edge set to "None" then
            // no more trees can be merged -- we are finished
            completed := true
        else
            completed := false
            for each component whose cheapest edge is not "None" do
                Add its cheapest edge to E'

function is-preferred-over(edge1, edge2) is
    return (edge2 is "None") or
           (weight(edge1) < weight(edge2)) or
           (weight(edge1) = weight(edge2) and tie-breaking-rule(edge1, edge2))

function tie-breaking-rule(edge1, edge2) is
    The tie-breaking rule; returns true if and only if edge1
    is preferred over edge2 in the case of a tie.
```

for each edge *uv* in *E*, where *u* and *v* are in different components of *F*:

A: let *wx* be the cheapest edge for the component of *u*

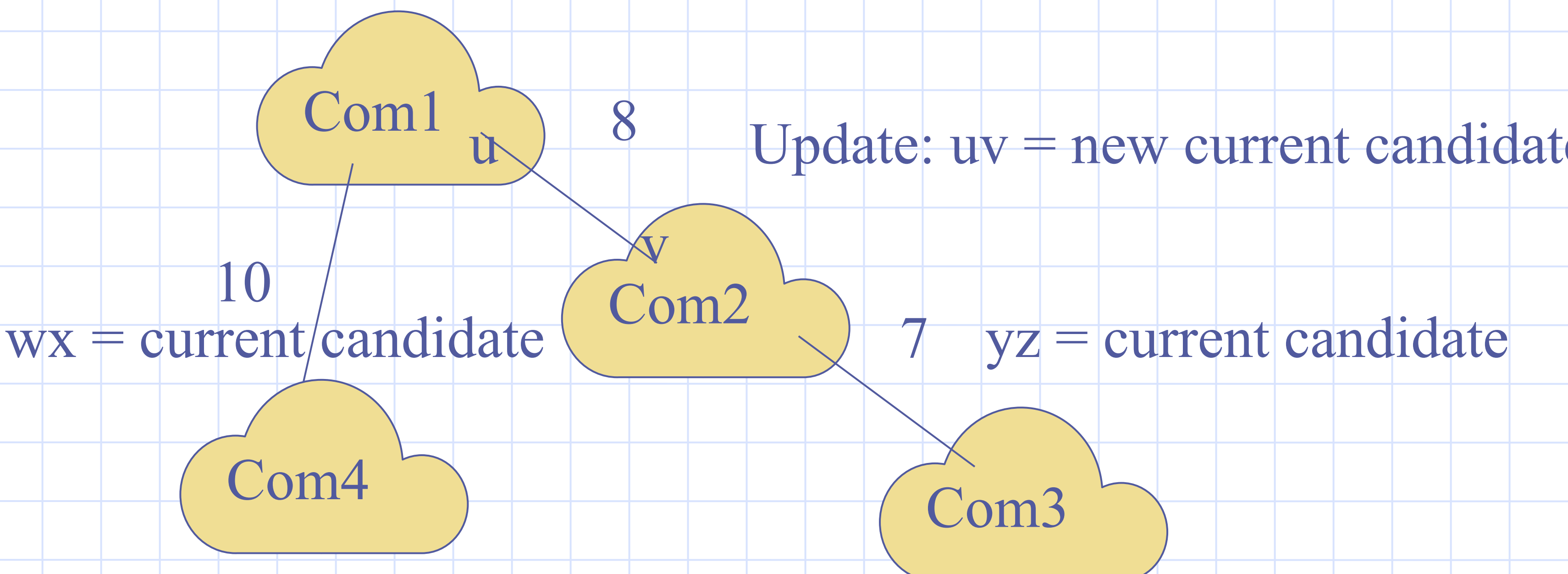
  if is-preferred-over(*uv*, *wx*) **then**

    Set *uv* as the cheapest edge for the component of *u*

B: let *yz* be the cheapest edge for the component of *v*

  if is-preferred-over(*uv*, *yz*) **then**

    Set *uv* as the cheapest edge for the component of *v*



```
algorithm Borůvka is
    input: A weighted undirected graph G = (V, E).
    output: F, a minimum spanning forest of G.

    Initialize a forest F to (V, E') where E' = {}.

    completed := false
    while not completed do
        Find the connected components of F and assign to each vertex its component
        Initialize the cheapest edge for each component to "None"
        for each edge uv in E, where u and v are in different components of F:
            let wx be the cheapest edge for the component of u
            if is-preferred-over(uv, wx) then
                Set uv as the cheapest edge for the component of u
            let yz be the cheapest edge for the component of v
            if is-preferred-over(uv, yz) then
                Set uv as the cheapest edge for the component of v
        if all components have cheapest edge set to "None" then
            // no more trees can be merged -- we are finished
            completed := true
        else
            completed := false
            for each component whose cheapest edge is not "None" do
                Add its cheapest edge to E'

function is-preferred-over(edge1, edge2) is
    return (edge2 is "None") or
           (weight(edge1) < weight(edge2)) or
           (weight(edge1) = weight(edge2) and tie-breaking-rule(edge1, edge2))

function tie-breaking-rule(edge1, edge2) is
    The tie-breaking rule; returns true if and only if edge1
    is preferred over edge2 in the case of a tie.
```

Trees, Forests and Mazes

- How to generate a Maze?
  - Think of a set of possible positions V in a space
  - Need a path from any position x to any other position y
  - Do not want too many paths or maze gets boring …
- Point of view:
  - Think of walls between all positions V
  - Each node is isolated and in its own set
  - If we remove a wall, we merge the sets the neighboring nodes belong to
  - If we remove enough walls such that all nodes belong to one set, we have a maze where all positions can be reached from another
- So:
  - Removing a wall is the same as adding an edge to a graph
  - Want a random maze? Pick random but unique weights

Summary

- Maze generation problem

- Seen as a graph problem
  - All nodes need to get connected
  - Randomized decisions which sets of nodes to connect / merge

- Boruvka's algorithm
  - Sequence of log n steps as in each step the # components cut in half
  - Each component merged with one of its adjacent components
    - Adjacent component select based on cheapest edge
  - Requires unique edge weights
    - For random maze: generate random but unique edge weights upfront