Algorithms and Data Structures

C4. Minimum Spanning Trees

Gabriele Röger and Patrick Schnider

University of Basel

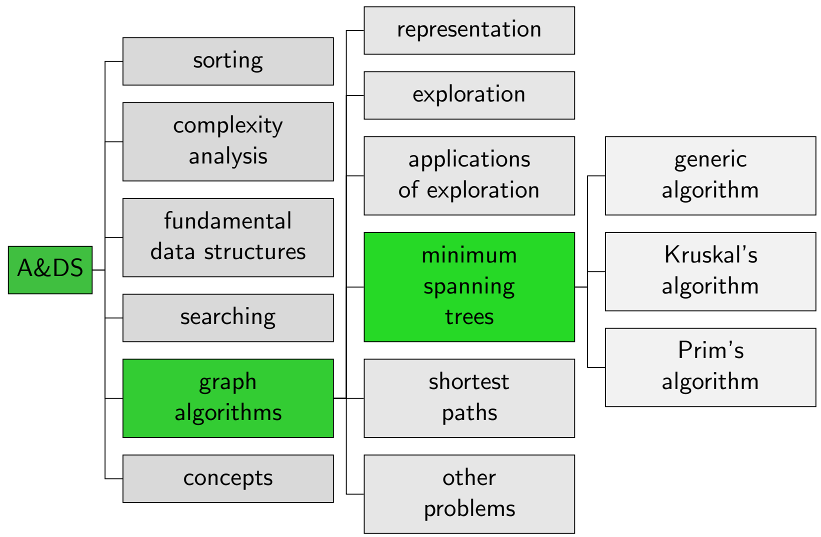
May 14, 2025

Algorithms and Data Structures

May 14, 2025 — C4. Minimum Spanning Trees

C4.1 Minimum Spanning Trees

Content of the Course

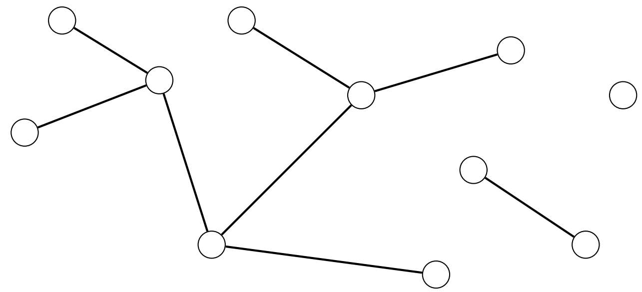


Undirected Graphs

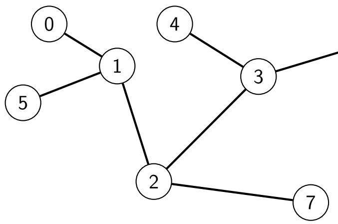
In chapter C4 we only consider undirected graphs.

Trees in Undirected Graphs

Definition
A tree is an acyclic connected graph.
A forest is an acyclic graph.



Properties of Trees

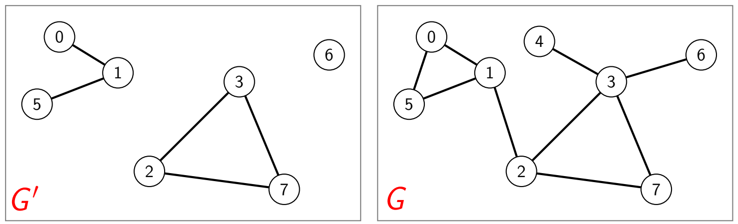


For every tree it holds that:

- Every pair of distinct vertices is connected by exactly one simple path (simple = no vertex occurs more than once).
- If we remove an edge, the graph becomes disconnected with two connected components.
- If we add an edge, we create a cycle.

Subgraph

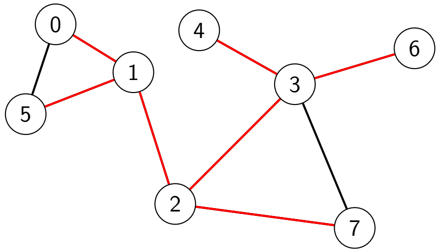
Definition
Graph $G' = (V', E')$ is a subgraph of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.



Spanning Tree

Definition

A spanning tree of a connected graph is a subgraph that contains all vertices of the graph and is a tree.



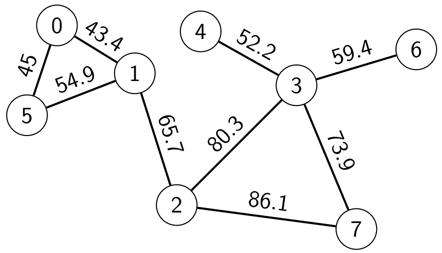
How many edges does a spanning tree have?

Weighted Graphs

Definition

An (edge-)weighted graph associates every edge e with a weight (or cost) $weight(e) \in \mathbb{R}$.

The weight of graph $G = (V, E)$ is the sum $weight(G) = \sum_{e \in E} weight(e)$ of its edge weights.



Minimum Spanning Trees

Definition (Minimum Spanning Tree Problem, MST Problem)

Given: Connected weighted undirected graph

Objective: Spanning tree with minimum weight (there is no spanning tree with a lower sum of edge weights).

Application: Clustering for Tumor Detection

Analysis of soft tissue tumors by an attributed minimum spanning tree.

Kayser K[1], Sandau K, Böhm G, Kunze KD, Paul J

Analytical and Quantitative Cytology and Histology [01 Oct 1991, 13(5):329-334]

Abstract

Histologic slides of 22 soft tissue tumors (9 malignant fibrous histiocytoma, 8 fibrosarcoma, 2 rhabdomyosarcoma, 2 osteosarcoma, 1 Askin tumor) were Feulgen stained. Using an automated image analyzing system (Cambridge 570) at low magnification (25x), the tumor cell nuclei were segmented. The geometrical center of the nuclei was considered the vertex. A basic graph was constructed according to the neighborhood condition of O'Callaghan. Neighboring tumor cell nuclei were visualized by connecting edges. Several features of tumor cell nuclei were measured, including area, surface, major and minor axis of best fitting ellipsis and extinction (DNA content). Nuclear features are attributed to the vertices. The differences, or "distances," between features of connected vertices are attributed to the corresponding edges, which are dependent on the attributes. Thus, different minimum spanning trees (MST) result. Each MST can be decomposed into clusters using a suitable decomposition function on the edges, which rejects an edge if its attributes differ from the mean of the attributed values of surrounding edges more than a neighbor dependent bound (lower limit). Taking into account the length and other attributes of edges (e.g., differences in orientation of the major axis), clusters of different nuclear orientation can be detected. A cluster tree can be constructed by defining the geometric center of a cluster as a new vertex, and by computing the neighborhood of the cluster vertices. The result is an attributed MST containing characteristic structural properties of the image (in cases of sarcomatous tumors, local orientation of tumor cell nuclei and local DNA abnormalities).

Application: Identity Recognition

Neurocomputing
Volume 72, Issues 7–9, March 2009, Pages 1859-1869

Minimum spanning tree based one-class classifier

Piotr Juszczak [a], David M.J. Tax [a], Elżbieta Pe̜kalska [b], Robert P.W. Duin [a]

⊞ Show more

https://doi.org/10.1016/j.neucom.2008.05.003

Abstract

In the problem of one-class classification one of the classes, called the target class, has to be distinguished from all other possible objects. These are considered as non-targets. The need for solving such a task arises in many practical applications, e.g. in machine fault detection, face recognition, authorship verification, fraud recognition or person identification based on biometric data.

This paper proposes a new one-class classifier, the minimum spanning tree class descriptor (MST_CD). This classifier builds on the structure of the minimum spanning tree constructed on the target training set only. The classification of test objects relies on their distances to the closest edge of that tree, hence the proposed method is an example of a distance-based one-class classifier. Our experiments show that the MST_CD performs especially well in case of small sample size problems and in high-dimensional spaces.

Application: Cell Nuclei Segmentation in Microscopy Images

Optimal cut in minimum spanning trees for 3-D cell nuclei segmentation

7 Author(s)
A. Abreu ; F.-X. Frenois ; S. Valitutti ; P. Brousset ; P. Denèfle ; B. Naegel ; C. Wemmert
View All Authors

Abstract | Authors | Figures | References | Citations | Keywords | Metrics | Media

Abstract:
In biology and pathology immunofluorescence microscopy approaches are leading techniques for deciphering of the molecular mechanisms of cell activation and disease progression. Although several commercial softwares for image analysis are presently in the market, available solutions do not allow a totally non subjective image analysis. There is therefore strong need for new methods that could allow a completely non-subjective image analysis procedure including for thresholding and for choice of the objects of interest. To address this need, we describe a fully automatic segmentation of cell nuclei in 3-D confocal immunofluorescence images. The method merges segments of the image to fit with a nuclei model learned by a trained random forest classifier. The merging procedure explores efficiently the fusion configurations space of an over-segmented image by using minimum spanning trees of its region adjacency graph.

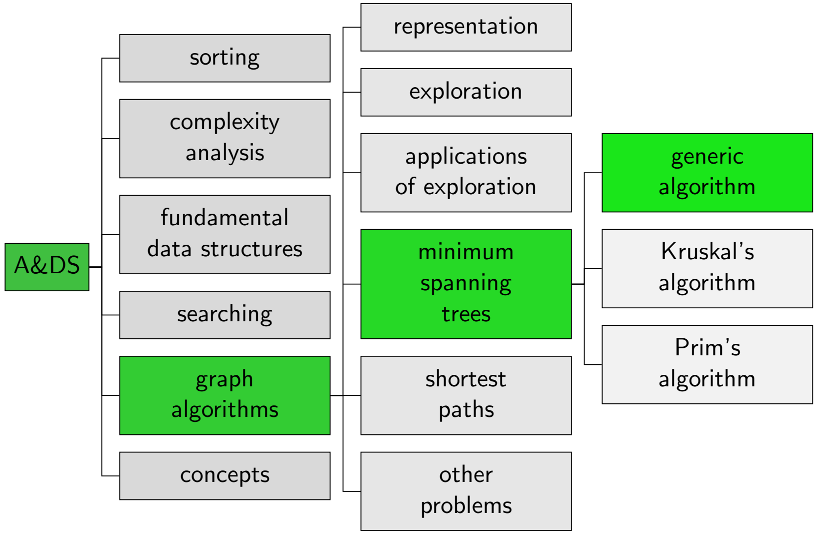
Published in: Image and Signal Processing and Analysis (ISPA), 2017 10th International Symposium on

Applications

- Network design
  - e.g. telecommunication networks, power networks
- Segmentation
  - e.g. of cell nuclei in microscopy images
- Cluster analysis
  - e.g. of cell nuclei for cancer diagnosis
- Approximation of hard graph problems
  - Steiner trees, Traveling Salesperson
- Many indirect applications
  - LDPC error-correcting codes
  - Features for face recognition
  - Ethernet protocol for avoiding cycles in broadcasting

C4.2 Generic Algorithm

Content of the Course



Generic Algorithm

For a subset A of the edges of a MST, we call edge e safe for A if $A \cup \{e\}$ is also a subset of the edges of a MST.

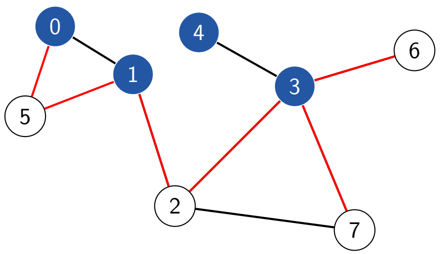
Input: Connected, undirected, weighted graph $G = (V, E)$

1. $A := \emptyset$
2. While (V, A) does not form a spanning tree of G:
   - Find an edge e that is safe for A.
   - $A = A \cup \{e\}$
3. Return (V, A)

Cuts in Graphs

Definition

Let $G = (V, E)$ be an undirected graph.
A cut $(V', V \setminus V')$ partitions the vertices.
An edge crosses the cut if one of its endpoints is in V' and the other endpoint in $V \setminus V'$.
The cut respects a set of edges $A \subset E$ if no $e \in A$ crosses the cut.



Sufficient Criterion for Safe Edges

Theorem

Let $G = (V, E)$ be a connected, undirected, weighted graph.

Let $A \subseteq E$ be a subset of the edges of some minimum spanning tree for G.

Let $(S, V \setminus S)$ be any cut of G that respects A and let e be an edge crossing the cut that has minimum weight among all such edges.

Then e is safe for A.

Sufficient Criterion for Safe Edges

Proof

Let T be a MST that includes A. If it includes e, we are done.

Otherwise we construct from T a MST T' that includes $A \cup \{e\}$.

Let u and v be the end points of e. The edge e forms a cycle with the edges on the simple path p from u to v in T.

Since e crosses the cut, path p must contain at least one edge that also crosses the cut. Let $e' = \{x, z\}$ be such an edge. Edge e' is not in A because the cut respects A.

Removing e' from T breaks it into two connected components. Adding e reconnects them into a new spanning tree T'. ...

Sufficient Criterion for Safe Edges

Proof (continued).

We still need to show that T' is a minimum spanning tree.

Since e is an edge of minimum weight among all edges that cross the cut and e' also crosses the cut, it holds that $weight(e) \leq weight(e')$. Therefore $weight(T') \leq weight(T)$.

Since T is a minimum spanning tree this implies that also T' is a minimum spanning tree.

The edges of T' include e and all edges from A (because $e' \notin A$), so overall we have shown that e is safe for A. □

Generic Algorithm

Input: Connected, undirected, weighted graph $G = (V, E)$

1. $A := \emptyset$
2. While (V, A) does not form a spanning tree of G:
   - Find an edge e that is safe for A.
   - $A = A \cup \{e\}$
3. Return (V, A)

- Why is there always a cut that respects A (as required by criterion for safe edges)?
- Terminates after $|V| - 1$ iterations. Why?
- Open question: How can we efficiently determine a safe edge?
  - Kruskal's algorithm
  - Prim's algorithm
- First: How do we represent the weighted graph?

C4.3 Graph Representation

Representation of Weighted Edges

Can extend previous representations:

- Adjacency matrix: Weight instead of binary entries
  - Can we support parallel edges?
- Adjacency list: Pairs of successor and weight in list.

But:

- Generic algorithm focuses on edges.
- Idea: Represent edges as objects.

API for Weighted Edge

```
class Edge:
  # edge between n1 and n2 with weight w
  def __init__(n1: int, n2: int, w: float) -> None

  # weight of the edge
  def weight() -> float

  # one of the two nodes
  def either_node() -> int

  # the other node (not n)
  def other_node(int n) -> int
```


Weighted Edge: Possible Implementation

```
class Edge:
    def __init__(self, n1, n2, weight):
        self.n1 = n1
        self.n2 = n2
        self.edge_weight = weight

    def weight(self):
        return self.edge_weight

    def either_node(self):
        return self.n1

    def other_node(self, n):
        if self.n1 == n:
            return self.n2
        return self.n1
```


Representation of Weighted Graphs

Graph representation

- We still want to be able to quickly determine the incident edges of a node.
- Store for every node references to the incident edges.
- Requires for every edge one object and two references to it.

API for Weighted Graphs

```
class EdgeWeightedGraph:
  # Graph with no_nodes nodes and no edges
  def __init__(no_nodes: int) -> None

  # add weighted edge
  def add_edge(e: Edge) -> None

  # number of nodes
  def no_nodes() -> int

  # number of edges
  def no_edges() -> int

  # all incident edges of node n
  def incident_edges(n: int) -> Generator[Edge]

  # all edges
  def all_edges() -> Generator[Edge]
```


Weighted Graph: Possible Implementation

```
class EdgeWeightedGraph:
    def __init__(self, no_nodes):
        self.nodes = no_nodes
        self.edges = 0
        self.incident= [[] for l in range(no_nodes)]

    def add_edge(self, edge):
        either = edge.either_node()
        other = edge.other_node(either)
        self.incident[either].append(edge)
        self.incident[other].append(edge)
        self.edges += 1

    def no_nodes(self):
        return self.nodes

    def no_edges(self):
        return self.edges
```


Weighted Graph: Possible Implementation (Continued)

```

    def incident_edges(self, node):
        for edge in self.incident_edges[node]:
            yield edge

    def all_edges(self):
        for node in range(self.nodes):
            for edge in self.incident_edges[node]:
                if edge.other_node(node) > node:
                    yield edge
```


API for MST Implementations

The algorithms for minimum spanning trees should implement the following interface:

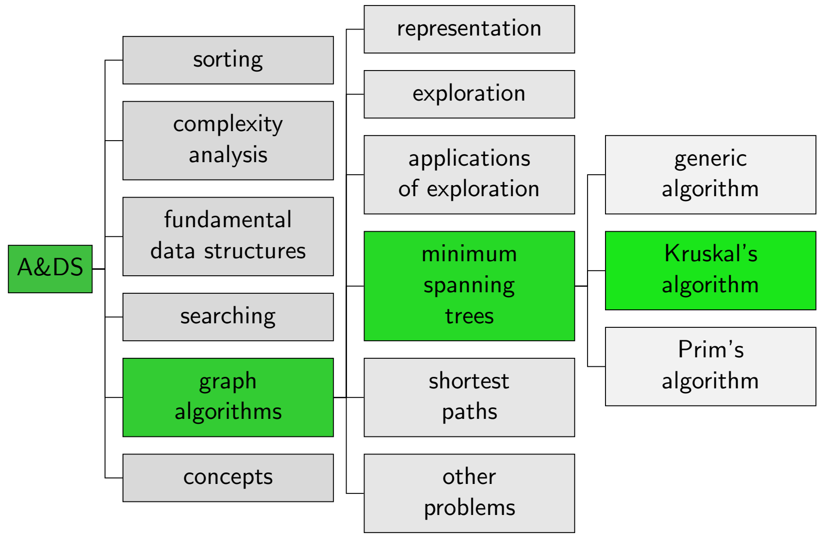
```
class MST:
  # initialization
  def __init__(graph: EdgeWeightedGraph) -> None

  # all edges of a minimum spanning tree
  def edges() -> Generator[Edge]

  # weight of the minimum spanning tree
  def weight() -> float
```

C4.4 Kruskal's Algorithm

Content of the Course



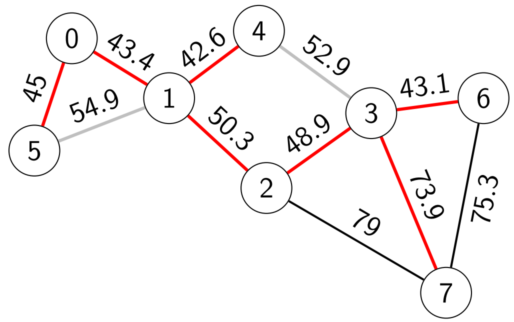
High-Level Perspective

Kruskal's Algorithm

- Process the edges in increasing order of their weights.
- Include the edge if it does not form a cycle with the already included edges. Otherwise, discard it.
- Terminate after including $|V| - 1$ edges.

Why is this an instantiation of the generic algorithm?

Illustration



red: included
gray: discarded

Kruskal's Algorithm Conceptually

Conceptional Approach

- Start with a forest of $|V|$ trees, where each tree only consists of a single node.
- Every included edge connects two trees into a single one.
- After $|V| - 1$ steps the forest consists of a single tree.

Questions

- How can we detect whether an edge connects two trees or whether both end points are in the same tree?
- Do we have to fully represent the individual trees?

→ We are only interested in the connected components

→ Disjoint sets to the rescue!

Kruskal's Algorithm: Implementation

```
class MSTKruskal:
    def __init__(self, graph):
        self.included_edges = []
        self.total_weight = 0
        candidates = minPQ()  # priority queue
        for edge in graph.all_edges():
            candidates.insert(edge)
        uf = UnionFind(graph.no_nodes())

        while (not candidates.empty() and
               len(self.included_edges) < graph.no_nodes() - 1):
            edge = candidates.del_min()
            v = edge.either_node()
            w = edge.other_node(v)
            if uf.connected(v, w):
                continue
            uf.union(v,w)
            self.included_edges.append(edge)
            self.total_weight += edge.weight()
```

How can methods edges() and weight() be implemented?

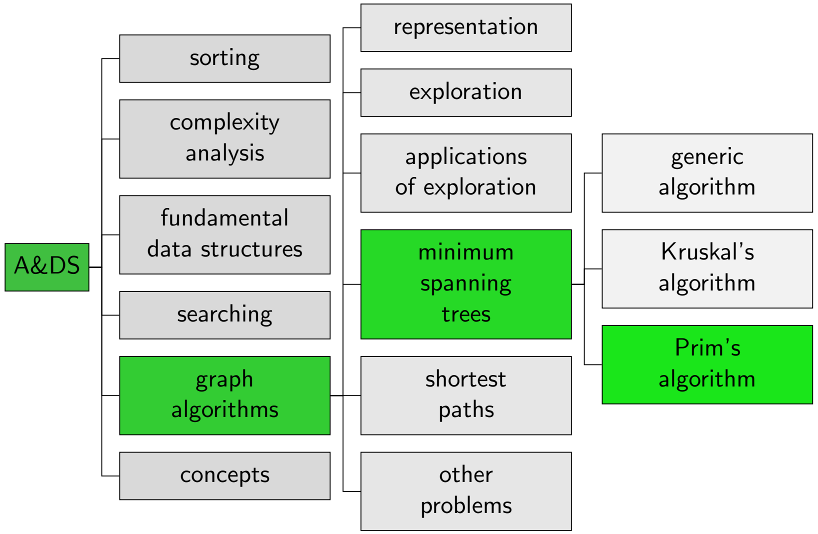
Kruskal's Algorithm: Running Time

- Assumption: Priority queue implemented as heap
- Initialization of priority queue with all edges: $|E|$ comparisons
- Never more than $|E|$ edges in the priority queue
  - Cost per operation is $O(\log_2 |E|)$
  - Total costs for priority queue operations is $O(|E| \log_2 |E|)$
- Dominates costs for union find structure.

In total: Running time $O(|E| \log_2 |E|)$, Memory $O(|E|)$

C4.5 Prim's Algorithm

Content of the Course



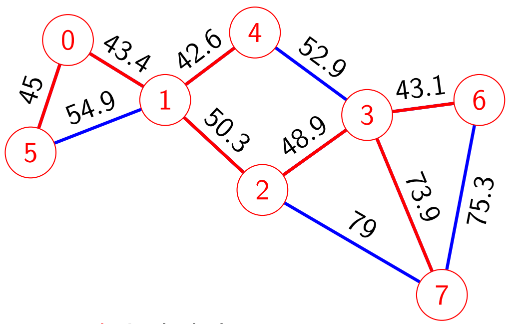
High-Level Perspective

Prim's Algorithm

- Choose an arbitrary node as initial tree.
- Let the tree grow by one additional edge in each step.
- Always add an edge of minimal weight that has exactly one end point in the tree.
  → safe edge
- Stop after adding $|V| - 1$ edges.

Illustration

With start vertex 0



red: included
blue: potential next edge

Implementation

Challenge:
Find the edge of minimal weight that has exactly one end point in the tree.

- Priority queue `candidates` that prioritizes edges by weight.
- Two variants:
  - eager: only edges that have exactly one endpoint are in the tree.
  - lazy: edges that have at least one end point in the tree

Main Loop of Lazy Variant

Invariant

Priority queue `candidate`

- contains all edges with exactly one endpoint in the tree
- and possibly edges with both endpoints in the tree.

While there are fewer than $|V| - 1$ added edges:

- Remove edge e with minimal weight from the priority queue.
- Discard e, if both end points in the tree.
- Otherwise, let v be the end point that is not yet in the tree.
  - Add all edges that are incident to v and whose other end point is not in the tree to `candidates`.
  - Add e and v to the tree.

Lazy Variant of Prim's Algorithm

```python
class LazyPrim:
    def __init__(self, graph):
        self.included_edges = []
        self.total_weight = 0

        # node-indexed list: True if node already in tree
        included_nodes = [False] * graph.no_nodes()
        candidates = minPQ()

        # include an arbitrary node (we use 0) in tree
        included_nodes[0] = True
        for edge in graph.incident_edges(0):
            candidates.insert(edge)
```


Lazy Variant of Prim's Algorithm (Continued)

```python

        while (not candidates.empty() and
               len(self.included_edges) < graph.no_nodes() - 1):
            edge = candidates.del_min()
            v = edge.either_node()
            w = edge.other_node(v)
            if included_nodes[v] and included_nodes[w]:
                continue
            if included_nodes[w]:
                v, w = w, v
            # v is in tree, w is not
            included_nodes[w] = True
            self.included_edges.append(edge)
            self.total_weight += edge.weight()
            for incident in graph.incident_edges(w):
                if not included_nodes[incident.other_node(w)]:
                    candidates.insert(incident)
```


Running Time and Memory

- Bottleneck is the number of comparisons of edge weights in methods `insert` and `del_min` of the priority queue.
- At most $|E|$ edges in priority queue
- Insertion and removal of minimum each take time $O(\log |E|)$
- At most $|E|$ insertions and $|E|$ removals
  $\rightarrow$ Running time $O(|E| \log |E|)$
- Memory $O(|E|)$

Eager Variant

Considerations

- We can remove edges from the priority queue if they already have both end points in the tree.
- If there are several edges that could connect a new node with the tree, we only can choose those of minimum weight.
- It is sufficient to always only consider one such edge.
- Idea: Remember one such edge for every node.
- The priority queue contains nodes, where the priority is the weight of the corresponding edge.

Problem: How can we efficiently update the priority queue?

Indexed Priority Queues

```
class IndexMinPQ:
  # Add key with priority val
  def insert(entry: Object, val: int) -> None

  # Remove and return entry with smallest priority
  def del_min() -> Object

  # Is the priority queue empty?
  def empty() -> bool

  # Does the priority queue contain the entry?
  def contains(entry: Object) -> bool

  # Change the priority of entry to val
  def change(entry: Object, val: int) -> None

  ...
```


Indexed Priority Queues

Priority queue implementation can easily be extended accordingly.

With a heap-based implementation we get running times

- $O(\log n)$ for insert, change and del_min
- $O(1)$ for contains and empty

Eager Variant of Prim's Algorithm: Data Structures

Do not use (indexed) priority queue of edges but

- edge_to: node-indexed array, containing at position v the edge (Edge) that connects v (in the direction of the start node) with the tree or could do so with the lowest weight.
- dist_to: Array containing at position v the weight of edge edge_to[v].
- pq: indexed priority queue of nodes
  - Nodes are not yet in the tree.
  - Can be connected by an edge with the existing tree.
  - Sorted by the weight of such an edge of lowest weight.

Eager Variant of Prim's Algorithm

```python
class EagerPrim:
    def __init__(self, graph):
        self.edge_to = [None] * graph.no_nodes()
        self.total_weight = 0
        self.dist_to = [float('inf')] * graph.no_nodes()
        self.included_nodes = [False] * graph.no_nodes()

        self.pq = IndexMinPQ()

        self.dist_to[0] = 0
        self.pq.insert(0, 0)
        while not self.pq.empty():
            self.visit(graph, self.pq.del_min())
```


Eager Prim-Algorithmus (Continued)

```python

    def visit(self, graph, v):
        self.included_nodes[v] = True
        for edge in graph.incident_edges(v):
            w = edge.other_node(v)
            if self.included_nodes[w]:
                continue
            if edge.weight() < self.dist_to[w]:
                # update cheapest edge between tree and w
                self.edge_to[w] = edge
                self.dist_to[w] = edge.weight()
                if self.pq.contains(w):
                    self.pq.change(w, edge.weight())
                else:
                    self.pq.insert(w, edge.weight())
```


Running Time and Memory

- Three node-indexed arrays
- At most $|V|$ nodes in the priority queue
- Memory $O(|V|)$
- Priority queue: need $|V|$ insertions, $|V|$ operations removing the minimum and at most $|E|$ changes of priority.
- Each operation possible in time $O(\log |V|)$.
- Running time $O(|E| \log |V|)$

C4.6 Outlook

Is there a MST Algorithm with Linear Running Time?

Algorithm	Memory	Running time
Kruskal	$\lvert E\rvert$	$\lvert E\rvert \log \lvert E\rvert$
Lazy Prim	$\lvert E\rvert$	$\lvert E\rvert \log \lvert E\rvert$
Eager Prim	$\lvert V\rvert$	$\lvert E\rvert \log \lvert V\rvert$
Fredman-Tarjan	$\lvert V\rvert$	$\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert$
Chazelle	$\lvert V\rvert$	$\lvert E\rvert \alpha(\lvert V\rvert)$ (almost $\lvert E\rvert$)
impossible?	$\lvert V\rvert$	$\lvert E\rvert$?

There is a randomized approach with expected linear running time [Karger, Klein, Tarjan, 1995].