Algorithmics

Sebastian Iwanowski
FH Wedel

4. Graph algorithms
4.1 Minimal spanning trees as motivation for basic algorithms

Algorithmics 4

4.1 Minimal spanning Trees

Kruskal's Algorithm (simple variant):

Construction of a minimum spanning tree for an arbitrary graph G:

- Start with an empty forest F consisting of no edge
- Repeat for all edges $e_1, e_2, \ldots, e_m$ of G (edges are in sorted order):
  Check if e_i may be inserted into F
  such that F is still without circles;
  If so, insert e_i into F;
  until F consists of n-1 edges (let n be the number of vertices of G).

Theorem: Thus constructed forest F is a minimum spanning tree of G

Proof: see next slide

How to do this smarter?

Time complexity: $O(m \log m + n^2)$ (nm due to determination of connectivity component)

References for catching up and delving into:

Skript Diskrete Mathematik 6, Folien 2,3,4,8,11,12,13 (graph theoretic basics)
Turau Kap. 2.4 (Grundlagen), 3.6.1 (Kruskal)
Cormen ch. 23 (Minimal spanning trees)

Algorithmik 4

4.1 Minimal spanning Trees

Proposition (implies correctness of Kruskal‘s algorithm, why?):

For each edge set $\{e_1, e_2, ..., e_j\}$ which is successively constructed by Kruskal‘s algorithm there is a minimum spanning tree T_j of G containing this edge set.

Proof by mathematical induction over j

Inductive step:

The assumption may hold for an edge set E_j consisting of j edges, i,e, there is a minimum spanning tree T_j where $E_j \subseteq T_j$.
Let e_{j+1} be the next edge chosen by Kruskal. If $k_{j+1} \in T_j$, choose $T_{j+1} = T_j$.
Otherwise there must be a circle in $T_j \cup \{e_{j+1}\}$ containing e_{j+1}. At least one of the other edges e_0 of this circle should not be contained in E_j (otherwise, Kruskal would not have chosen e_{j+1} because E_j would not have been free of circles). Replace this edge e_0 by edge w_{j+1} => spanning tree T_{j+1} containing $E_j \cup \{e_{j+1}\}$.
$c(e_0) \geq c(e_{j+1})$, because otherwise Kruskal would have chosen e_0 before e_{j+1}.
Thus, T_{j+1} must be minimum as well as T_j.

Deutschsprachige Referenzen zum Nacharbeiten und Vertiefen:

Skript Alt, Lemma 4.3.2 (S. 76): Beweisskizze eines verwandten Satzes
Turau, Kapitel 3.6.1: genauer Beweis des Satzes wie oben (inkl. Induktionsverankerung)
Lang: Skript Berechenbarkeit und Komplexität, Kap. 4.2.3 (Greedy-Algorithmen für Matroide)

Algorithmics 4

4.1 Basic algorithms for graph theory

Union-Find-Structure

In general: works on sets of sets,
implements efficient location of the set of a given element
and efficient union of sets

Graph theoretic application: efficient location and union of connectivity components

O(log n)	`Find (v)`	returns a unique reference node of the connectivity component of v.
O(1)	`Union (v,w)`	unifies the connectivity components of v and w after reference node has been determined

With path compression:

Expected time complexity of Find is in O(log*n)

Data representation:

Array of nodes: The contents are pairs of the form (index of parent, height of subtree)

References:

Skript Alt, Kap. 3.2 (p. 56 ff.), Cormen ch. 21 (Data structures for disjoint sets)

Algorithmics 4

4.1 Basic algorithms for graph theory

Heap

Efficient management of a priority queue

Invariants:
1) A heap is a complete binary tree (elements may be missing only in the last depth level).
2) The keys of the children of each node are not less than the key of each node.

O(log n)	**`DeleteMin()`**	deletes the minimal element of the heap.
O(log n)	**`Insert (v)`**	inserts an arbitrary new element into the heap.
O(1)	**`SearchMin()`**	finds the minimal element of the heap.

Data representation:

Array of the heap nodes:

The contents are the contents of the heap nodes.
The children of the node with index i are the nodes with indices 2i und 2i+1
(assuming that the array starts with index 1)

References:

Cormen, ch. 6 (Heapsort)

Algorithmics 4

4.1 Minimal spanning Trees

Kruskal‘s Algorithm (efficient variant):

Construction of a minimal spanning tree for an arbitrary graph G:

- Start with an empty forest F consisting of no edge
- Start with a **union-find-structure** in which each vertex has its own connectivity component
- Insert all edges into a **heap**

- While F consists of less than n-1 edges:
  Search and delete the minimal element e_{min} from the heap;
  Check if the vertices v and w incident with e_{min} are in the same connectvity component
  If not: Insert e_{min} into F and unify the connectivity components of v and w.

Time complexity: O(m log m) (m is the number of edges in G)

References:

Cormen, ch. 23.2 (Algorithms of Kruskal and Prim)