LEDA

Relevante Klassen und Befehle zum Arbeiten mit Graphen und SSSP-Algorithmen

Klassen, Funktionen	Beschreibung	Infos
class graph	*#include <LEDA/graph.h>*	
graph G; node v; edge e;	• legt eine Instanz der Klasse *graph* an • Typ: *node* (Typ, um später auf Knoten zugreifen zu können) • Typ: *edge* (Typ, um später auf Kanten zugreifen zu können)	
node v = G.new_node(); edge e = G.new_edge(v1, v2);	• erzeugt einen Knoten in G; später Zugriff auf den Knoten möglich • erzeugt eine gerichtete Kante von v1 zu v2; später Zugriff möglich	
class GRAPH	*#include <LEDA/graph.h>*	
GRAPH<T1,T2> G; G.new_node(wert<T1>); G.new_edge(v1,v2,wert<T2>);	Abgeleitet von *graph* für parametrisierte und dynamische Graphen, bietet Knoten und Kanten Platz für je eine weitere Information, die Parameter werden direkt in G gespeichert.	schnell; dynamisch
G[e oder v] = wert<T1 bzw. T2> G.inf(v oder e);	• Beispiel für eine Zuweisung (lesen + schreiben) • mit inf kann nur gelesen werden	
Funktionen (graph / GRAPH)		
G.number_of_nodes(); G.number_of_edges();	• liefert die Zahl aller Knoten des Graphen G • liefert die Zahl aller Kanten des Graphen G	
node_array<T> name(G); edge_array<T> name (G,x); name[v oder e] = Wert<T>;	• array, der die Knoten des vollendeten Graphen G parametrisiert • array, der die Kanten des vollendeten Graphen G parametrisiert [*(G)*: default, *(G,x)*: mit x initialisieren, *(G,n,x)* reserviert für n Knoten bzw. Kanten Speicher, falls G noch wachsen sollte]	mittel; nicht dynamisch
graph G(n_slots, e_slots); node_array<T> name; name.use_node_data(G, [n,x]);	Falls im Vorfeld schon bekannt ist, dass G *n_slots node_arrays* und *e_slots edge_arrays* haben wird (diese Konstruktion ist schneller).	mittel bis schnell
G.sort_nodes(node_array); G.sort_edges(edge_array);	Sortiert die in G gespeicherte Abfolge der Knoten bzw. Kanten nach den Werten, wie sie in dem übergebenen array gespeichert sind.	
G.out_edges(v); G.in_edges(v); G.adj_edges(v);	Drei Listen, die jeder Knoten eines gerichteten Graphen besitzt. Darin sind alle kommenden und gehenden (gerichteten) Kanten gespeichert (*adj_edges* = *out_edges*).	
G.make_undirected(); G.make_directed(); G.make_bidirected();	• *out_edges(v)* und *in_edges(v)* werden in *adj_edges(v)* vereinigt • richtet G wieder (nicht mehr identisch mit dem Ursprungsgraphen) • ergänzt den Graphen um Kanten, damit er bidirektional wird	
node s = G.source(e); node t = G.target(e); node o = G.opposite(e,v);	• liefert den Quellknoten an der gerichteten Kante e; • liefert den Zielknoten an der gerichteten Kante e; • liefert den zweiten Knoten an der Kante e (v ist der erste Knoten)	
G.outdeg(v); G.indeg(v); G.degree(v);	• der Ausgangsgrad eines Knotens (Zahl der abgehenden Kanten) • der Eingangsgrad eines Knotes (Zahl der eingehenden Kanten) • Ausgangs- + Eingangsgrad	
G.print_node(v); G.print_edge(e);	• gibt die Nummer des Knotens v in eckigen Klammern aus: [#] • gibt die Nummer der Kante e in eckigen Klammern aus: [#]	

Iteratoren (Makros)		
forall_nodes(v,G) {Quellcode}; forall_edges(e,G) {Quellcode}; forall_rev_nodes(v,G) {Qc.}; forall_rev_edges(e,G) {Qc.};	Diese Makros werden wie Schleifen behandelt, durchlaufen alle Kanten bzw. Knoten von G und speichern sie bei jedem Durchlauf in v bzw. e ab. Mit *rev* werden die gespeicherten Knoten bzw. Kanten rückwärts durchlaufen.	
forall_adj_nodes(v,w) {Qc.}; forall_adj_edges(e,w) {Qc.};	• nodes: G gerichtet: Durchläuft alle v, von denen w die Wurzel ist; • nodes: G unger.: Alle Knoten v, die eine Kante zu w besitzen; • edges: G gerichtet: Findet alle e, deren Quelle der Knoten w ist; • edges: G ungerichtet: Durchläuft generell alle e am Knoten w;	mittel bis schnell
forall_out_edges(e,w) {Qc.}; forall_in_edges(e,w) {Qc.}; forall_inout_edges(e,w) {Qc.};	• schnellere Version von *forall_adj_edges()*; • iteriert über alle Kanten, deren Ziel w ist; • iteriert zuerst über alle Kanten in *out_edges()*, dann in *in_edges()*;	schnell
Speichern, Laden, Erzeugen	*#include <LEDA/graph_gen.h>* (zum Generieren von Graphen)	
G.write(gw-datei<string>); test = G.read(gw-datei<string>); G.write_gml(gml-datei<string>); G.read_gml(gml-datei<string>);	• Speichern und Lesen eines Graphen G in einer gw-Datei; • test: 0=ok; 1=Datei nicht gefunden; 2=fehlerhaft; 3=kein Graph; • Schreibt im gml-Format (XML) • Liest das gml-Format (XML)	
complete_graph(G<&>, n<int>);	• erzeugt G mit n Knoten und n-1 Kanten, alle Knoten verbunden	
random_graph(G,n,m,x,y,z);	• n Knoten, m Kanten; (bool x,y,z): (keine Rückwärtskanten [true], keine Schleifen [true], keine Parallelkanten [true])	
LEDA Tools für SSSP		
edge_array<int> cost(G); node_array<edge> pred(G); node_array<int> dist(G); SHORTEST_PATH(G,n0,cost, dist,pred);	*#include <LEDA/shortest_path.h>* Ein SSSP-Algorithmus aus LEDA, mit Graph G und Startknoten *n0*; *cost* übergibt die Kosten der einzelnen Kanten; *pred* speichert die letzte Kante vor dem Zielknoten v; *dist* speichert die Gesamtdistanz (Kosten) von n0 nach v	O(x); x= |V|*log |V|+|E|
edge_array<int> cost(G); node_array<edge> pred(G); node_array<int> dist(G); DIJKSTRA(G,n0,cost, dist,pred);	*#include <LEDA/shortest_path.h>* siehe Beschreibung zu SHORTEST_PATH();	O(x); x= |V|*log |V|+|E|
node_array<int> test(G); CHECK_SP(G,n0,cost,dist,pred);	*#include <LEDA/shortest_path.h>* wird nach SSSP ausgeführt; *test* liefert für jeden untersuchten Knoten v: test[v] > 0: v ist von n0 nicht erreichbar; <= 0: erreichbar	O(x); x= |V|+|E|
DIJKSTRA_T<T> (…); SHORTEST_PATH_T<T> (…); CHECK_SP_T<T> (…);	*#include <LEDA/templates/shortest_path.t>* Verwendet obige Funktionen als Template-Funktionen, falls die Kanten des Graphen nicht mit *int* parametrisiert sind.	

Diese Zusammenfassung legt ihren Schwerpunkt nicht auf Vollständigkeit, sondern dient als Hilfe beim Programmieren mit Graphen in LEDA.

Es empfiehlt sich, SSSP-Algorithmen mit dem gleichen Interface wie SHORTEST_PATH(); und DIJKSTRA(); zu implementieren.