Einführung in die Programmierung mit C++

Dynamische Datenstrukturen (Graphen)

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Graphen

- Knoten
- Kanten
- Konstruktion
- Destruktion
- Verwendung

Tiefensuche auf Graphen

- Besucher
- Generik

Outlook: Boost [Graph Library]

Outline

Graphen

Definition

Betrachte Graphen $G = (V, E)$ mit Knoten V und Kanten $E \subseteq V \times V$.

```
1  #include<vector>
2  #include<list>
3
4  template<typename VT, typename ET>
5  class graph {
6    std::vector<VT*> vertices;
7    std::list<ET*> edges;
8  public:
9    graph(char*);
10   ~graph();
11   void to_dot() const;
12 private:
13   graph();
14   graph(const graph&);
15   graph& operator=(const graph&);
16 };
17
18 #include "graph.cpp"
```

- Ein Vektor von Zeigern auf Knoten variablen Typs (VT) ermöglicht Identifizierung dieser über die Position innerhalb des Vektors und schnellen Zugriff via vertices[i].
- Eine Liste von Zeigern auf Kanten variablen Typs (ET) ermöglicht effizientes Einfügen und Entfernen.
- Der Graph wird bei Konstruktion aus einer Datei eingelesen. Die Ausgabe auf den Bildschirm erfolgt im `graphviz`-Format.
- Weder die Konstruktion leerer Graphen, noch deren Kopieren bzw. Zuweisen (*deep copy*) sollen möglich sein.

Knoten sind Spezialisierungen der Klasse indexed, welche die Zuweisung eines eindeutigen Index zu jedem Knoten implementiert.

```
1  #include "indexed.hpp"
2
3  class vertex : public indexed {
4  public:
5    vertex()=default;
6  private:
7    vertex(const vertex&);
8    vertex& operator=(const vertex&);
9  };
```

- Die Konstruktion von Knoten reduziert sich auf die Konstruktion des jeweiligen Basisobjekts vom Typ indexed durch den Standardkonstruktor. Eine Implementierung dessen wird durch den Compiler automatisch generiert.
- Weder Kopieren noch Zuweisen von Knoten soll möglich sein, d.h. Knoten sind eindeutig.

Indizierte Objekte erlauben eindeutige Identifizierung über einen entsprechenden Index.

```
1  #include <cstddef>
2
3  class indexed {
4    static size_t counter;
5    size_t i;
6  public:
7    indexed();
8    size_t index() const;
9  private:
10   indexed(const indexed&);
11   indexed& operator=(const indexed&);
12 };
13
14 size_t indexed::counter=0;
15
16 indexed::indexed() : i(counter++) {}
17
18 size_t indexed::index() const { return i; }
```

- Die Eindeutigkeit des Index i ist durch den Instanzenzähler counter garantiert. Hier stimmen Index eines Knotens und dessen Position im durch graph verwalteten Vektor von Zeigern auf Knoten überein (ohne Einschränkung der Allgemeinheit).
- Der Lesezugriff auf den Index erfolgt über index().
- Indizierte Objekte sollen eindeutig sein, d.h. sie dürfen weder kopiert noch zugewiesen werden.

Kanten sind Paare adjazenter Knoten.

```
1  template<typename VT>
2  class edge {
3  public:
4    VT *src, *tgt;
5    edge(VT* s, VT* t);
6  private:
7    edge();
8    edge(const edge&);
9    edge& operator=(const edge&);
10 };
11
12 template<typename VT>
13 edge<VT>::edge(VT* s, VT* t)
14   : src(s), tgt(t) {}
```

- Zeiger auf die zur Kante inzidenten Endknoten werden gespeichert und müssen zum Zeitpunkt der Konstruktion der Kante spezifiziert werden.
- Weder die Konstruktion leerer Kanten (ohne Endknoten), noch deren Kopieren bzw. Zuweisen möglich sein.

Graphen werden bei Konstruktion aus einer Datei eingelesen.

```
1  template<typename VT, typename ET>
2  graph<VT,ET>::graph(char* filename) {
3    std::ifstream ifs(filename);
4    int n,m; // numbers of vertices and edges
5    ifs >> n >> m;
6    // insert vertices
7    for (int i=0;i<n;i++)
8      vertices.push_back(new VT);
9    // insert edges
10   int src, tgt;
11   for (int i=0;i<m;i++) {
12     ifs >> src >> tgt;
13     edges.push_back(
14       new ET(vertices[src],vertices[tgt]));
15   }
16 }
```

- Valide Dateien enthalten die Anzahlen von Konten (n) und Kanten (m) gefolgt von m Paaren von Knotenindizees, welche die entsprechenden Kanten repräsentieren.
- n Knoten werden alloziert und deren Adressen werden dem entsprechenden Vektor sukzessive hinzugefügt.
- Beim Hinzufügen der m Kanten erfolgt keine (wünschenswerte) Überprüfung der Zulässigkeit der jeweiligen Knotenindizees (0<=src|tgt<n).

Die Ausgabe des Graphen erfolgt im `graphviz`-Format auf den Bildschirm (Details im Quelltext).

```
1 template<typename VT, typename ET>
2 void graph<VT,ET>::to_dot() const {
3   ...
4 }
```

Bei Destruktion des Graphen werden sowohl Knoten als auch Kanten dealloziert.

```
1 template<typename VT, typename ET>
2 graph<VT,ET>::~graph() {
3   for (auto &v : vertices) delete v;
4   for (auto &e : edges) delete e;
5 }
```

Graphen mit Knoten und Kanten variablen Typs können basierend auf der gegebenen strukturellen Spezifikation (Datei) konstruiert und im `graphviz`-Format auf den Bildschirm ausgegeben werden.

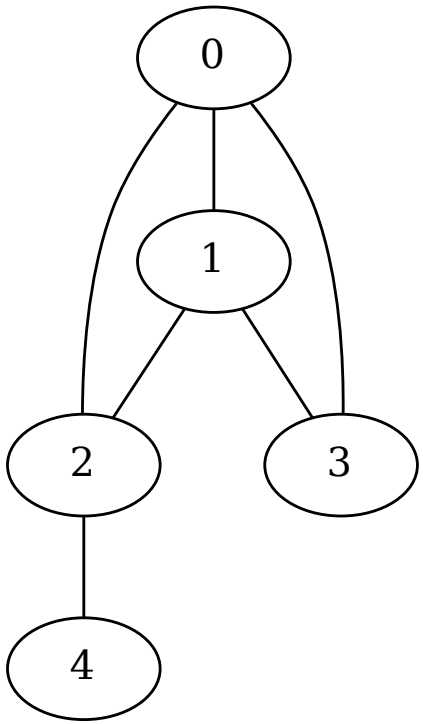
```
1  #include "vertex.hpp"
2  #include "edge.hpp"
3  #include "graph.hpp"
4
5  #include <cassert>
6
7  int main(int argc, char* argv[]) {
8    assert(argc==2);
9    graph<vertex,edge<vertex>> g(argv[1]);
10   g.to_dot();
11   return 0;
12 }
```

- Graphen, Knoten und Kanten sind in den jeweiligen *header*-Dateien definiert.
- Graphen und Kanten sind *templates*, Knoten nicht.
- Name der Spezifikationsdatei wird als einziger (assert) Kommandozeilenparameter erwartet und an den Konstruktor des Graphen übergeben.

Eine Spezifikationsdatei der Form

```
5
6
0 1
0 2
0 3
1 2
1 3
2 4
```

resultiert in folgendem Graph:



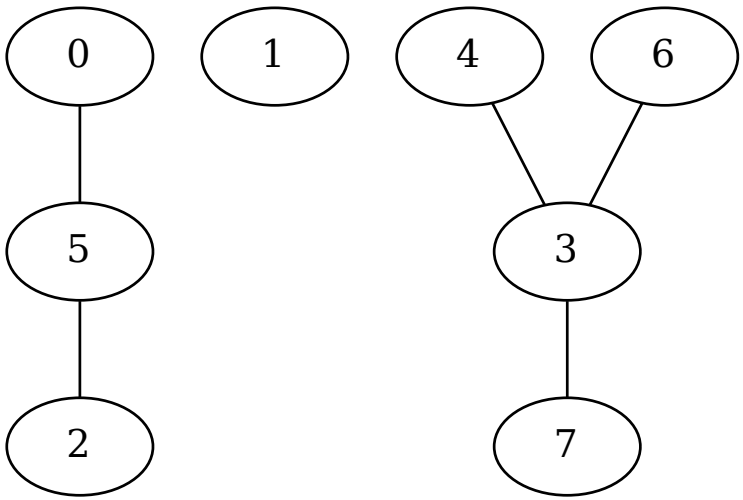
In der aktuellen Form ist der Graph noch keine gute Grundlage für die effiziente algorithmische Verarbeitung (z.B. Bestimmen aller Nachbarn eines Knotens).

Outline

Graphen können aus mehreren untereinander nicht verbundenen Teilgraphen (Komponenten) bestehen.

Für einen gegebenen Knoten kann die Komponente, zu welcher er gehört, durch Suche nach allen Knoten, die vom gegebenen über einen Pfad (Sequenz paarweise adjazenter Knoten) erreichbar sind, bestimmt werden.

```
8
5
0 5
5 2
3 7
6 3
4 3
```



Nebenstehender Graph hat drei Komponenten. Knoten in verschiedenen Komponenten sind untereinander nicht erreichbar, z.B. 2,1,7.

Alle Knoten, welche zur selben Komponente wie ein gegebender Knoten gehören, können z.B. mittels Tiefensuche bestimmt werden.

Tiefensuche auf Graphen

(indizierte, zu besuchende) Knoten


```
1  #include "identifiable.hpp"
2  #include "visitable.hpp"
3
4  #include <list>
5
6  template<typename VT>
7  class edge;
8  template<typename VT, typename ET>
9  class graph;
10
11 class vertex :
12       public identifiable<size_t>,
13       public visitable {
14    std::list<edge<vertex>*> incident;
15 public:
16    vertex()=default;
17 private:
18    vertex(const vertex&);
19    vertex& operator=(const vertex&);
20    friend graph<vertex,edge<vertex>>;
21 };
```

Knoten werden um ihre Nachbarschaft erweitert $\Rightarrow$ effizientere Algorithmen auf Kosten zusätzlichen Speicherbedarfs.

- Identifizierbarkeit und Besuchbarkeit von Knoten durch multiple Vererbung
- Knoten speichern Zeiger auf inzidente Kanten $\Rightarrow$ effiziente Bestimmung der Nachbarschaft
- Graph erhält Zugriff auf Nachbarschaft $\rightarrow$ **friend** :-(
- Vorwärtsdeklaration von `edge` und `graph`

Tiefensuche auf Graphen

Besuchbarkeit (von Knoten)


```
1  class visitable {
2    bool v=false;
3  public:
4    visitable()=default;
5    const bool& visited() const;
6    bool& visited();
7  private:
8    visitable(const visitable&);
9    visitable& operator=(const visitable&);
10 };
11
12 const bool& visitable::visited() const {
     return v; }
13
14 bool& visitable::visited() { return v; }
```

- Wiederholte Tiefsuche für denselben Knoten sollte aus Effizienzgründen vermieden werden.
- Zu diesem Zweck wird für initial nicht besuchte Knoten bei Besuch der Wert von `v` auf **true** gesetzt.
- Entsprechende Routinen für Lese- und Schreibzugriff werden zur Verfügung gestellt.
- Besuchbare Objekte sollen eindeutig sein, d.h. sie dürfen weder kopiert noch zugewiesen werden.

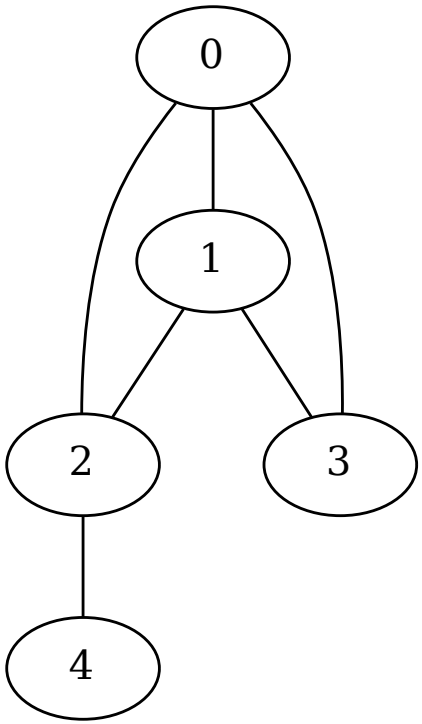
Tiefensuche auf Graphen

Identifizierbarkeit (von Knoten)


```
1  template<typename T>
2  class identifiable {
3    static T counter;
4    T i;
5  public:
6    identifiable();
7    T identifier() const;
8  private:
9    identifiable(const identifiable&);
10   identifiable& operator=(const identifiable&);
11 };
12
13 // required: T(size_t)
14 template<typename T>
15 T identifiable<T>::counter=T(0);
16
17 // required: T::operator++
18 template<typename T>
19 identifiable<T>::identifiable() : i(counter++) {}
20
21 template<typename T>
22 T identifiable<T>::identifier() const { return i; }
```

- Knoten sind über eine Variable i eindeutig identifizierbar.
- Lesezugriff wird gewährt.
- Deren Typ muss zu Null initialisierbar und inkrementierbar sein.
- Inkremetierung einer statischen "Zählervariable" erfolgt bei Konstruktion.
- Identifizierbare Objekte sollen eindeutig sein, d.h. sie dürfen weder kopiert noch zugewiesen werden.

```
5
6
0 1
0 2
0 3
1 2
1 3
2 4
```



Es sei die Nachbarschaft der Knoten gemäß der Konstruktion ihrer inzidenten Kanten geordnet, d.h. z.B. ist 1 erster Nachbar von 0 und 2 sowie zweiter Nachbar von 3.

Tiefensuche (DFS; *depth first search*) von 0 ausgehend besucht 1 (über Kante (0,1)), 2 (1,2), 4 (2,4) und 3 (1,3) in dieser Reihenfolge. Jeder Knoten wird einmal besucht.

Tiefensuche von 3 ausgehend besucht 0 (0,3), 1 (0,1), 2 (1,2) und 4 (2,4) in dieser Reihenfolge.

Tiefensuche auf Graphen

Implementierung

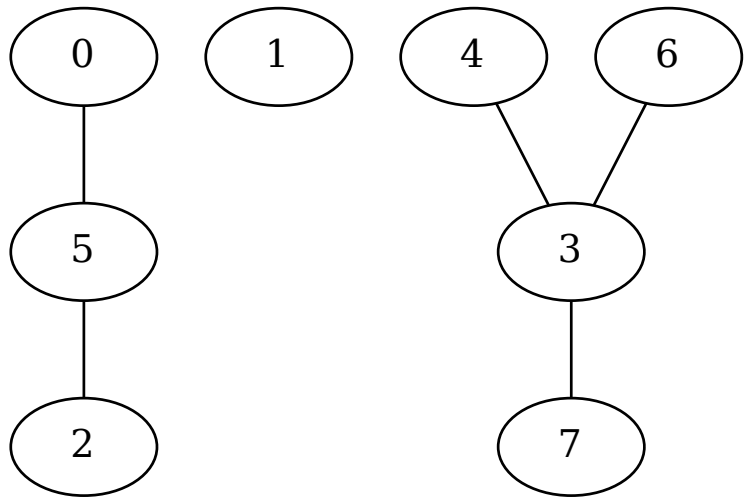

```
1  template<typename VT, typename ET>
2  class graph { ...
3  public:
4     ...
5     void dfs(VT&) const;
6     void dfs(VT&,void(*)(VT&)) const;
7     ...
8  };
9
10 template<typename VT, typename ET>
11 void graph<VT,ET>::dfs(VT& v) const {
12    if (v.visited()) return;
13    v.visited()=true;
14    std::cerr << v.identifier() << std::endl;
15    for (ET*& e : v.incident)
16       if (e->src==&v)
17          dfs(*(e->tgt));
18       else
19          dfs(*(e->src));
20 }
```

- Zwei Varianten: Ausgabe der Knoten-ID; generischer Besucher.
- Initial unbesuchte Knote werden als “besucht” markiert.
- Bereits besuchte Knoten unterbrechen die Rekursion.
- Rekursiver Besuch aller adjazenten Knoten gemäß ihrer Reihenfolge innerhalb der Nachbarschaft.
- In einem ungerichteten Graphen müssen beide Endknoten einer Kante behandelt werden.

Tiefensuche auf Graphen

Fallstudie


```
8
5
0 5
5 2
3 7
6 3
4 3
```




```
dfs(0): 0 5 2
dfs(1): 1
dfs(2): 2 5 0
dfs(3): 3 7 6 4
dfs(4): 4 3 7 6
dfs(5): 5 0 2
dfs(6): 6 3 7 4
dfs(7): 7 3 6 4
```

```
1  void print(const vertex& v) {
2    std::cerr << v.identifier() << std::endl;
3  }
4
5  template<typename VT, typename ET>
6  void graph<VT,ET>::dfs(VT& v, void (*f)(
       const VT&)) const {
7    if (v.visited()) return;
8    v.visited()=true;
9    f(v);
10   for (ET*& e : v.incident)
11     if (e->src==&v)
12       dfs(*(e->tgt),f);
13     else
14       dfs(*(e->src),f);
15 }
```

- Übergabe einer (auf Knoten agierenden) Funktion an dfs, z.B. Ausgabe der Knoten-ID; alternativ: Ausgabe der Adresse des Knotens im Speicher etc.
- Logik des Algorithmus bleibt unverändert.
- Modernes C++ bietet bessere Alternativen zur Implementierung der Generik → Advanced C++

Outline

Boost [Graph Library]

Outlook


```
1  #include <boost/graph/undirected_graph.hpp>
2  #include <boost/graph/graphviz.hpp>
3
4  int main() {
5    boost::undirected_graph<> g;
6
7    auto v0 = g.add_vertex();
8    auto v1 = g.add_vertex();
9    auto v2 = g.add_vertex();
10
11   g.add_edge(v0, v1);
12   g.add_edge(v1, v2);
13
14   write_graphviz(std::cout, g);
15
16   return 0;
17 }
```

- `www.boost.org`: "Boost provides free peer-reviewed portable C++ source libraries."
- Beispiele: Gleitkommazahlen beliebiger Genauigkeit, Intervalle, Graphen, ...; Oft ist die Zugehörigkeit zu Boost Indikator für spätere Aufnahme in den C++ Standard, z.B. Threads, schlaue Zeiger.
- Die Boost Graph Library bietet weitreichende Unterstützung für Graphen und entsprechende Algorithmen.

Zusammenfassung

Graphen

- Knoten
- Kanten
- Konstruktion
- Destruktion
- Verwendung

Tiefensuche auf Graphen

- Besucher
- Generik

Outlook: Boost [Graph Library]