Graphen

Wolfgang Mulzer

1 Operationen

Der ADT Graph bietet folgende Operationen:

- `vertices`(): liefere einen Iterator für die Knoten;
- `edges`(): liefere einen Iterator für die Kanten;
- `createVertex`(): erzeuge einen Knoten;
- `deleteVertex`(v): lösche Knoten v;
- `createEdge`(u, v): erzeuge Kante zwischen u und v (unterschiedliche Semantik für gerichtete und ungerichtete Graphen);
- `deleteEdge`(e): lösche Kante e.

Operationen für einen Knoten v:

- v.`incidentEdges`(): liefere einen Iterator für die inzidenten Kanten. In gerichteten Graphen gibt es zwei Operationen: `outgoingEdges`() und `incomingEdges`().
- v.`setInfo`(o): speichere das Objekt o am Knoten v;
- v.`getInfo`(): liefere das Objekt am Knoten v.

Operationen für eine Kante e:

- e.`endVertices`(): liefere die beiden Endknoten von e;
- e.`opposite`(v): liefere den Endknoten von e, der nicht v ist:
- e.`setInfo`(o): speichere das Objekt o an der Kante e;
- e.`getInfo`(): liefere das Objekt an der Kante e.

2 Tiefensuche rekursiv

Jeder Knoten v hat ein `boolean`-Attribut v`.found`, das anzeigt, ob v von der DFS schon entdeckt wurde. Zu Beginn ist v`.found` = `false` für alle Knoten $v \neq s$, und s`.found` = `true`. Dabei ist s der Startknoten. Der Aufruf erfolgt durch `dfs`(s).

```
dfs(v)
  //process v
  // Besuche rekursiv alle Nachbarn, die noch
  // nicht entdeckt wurden
  for e in v.outgoingEdges() do
    w <- e.opposite(v)
    if not w.found then
      w.found <- true
      dfs(w)
```

3 Tiefensuche iterativ

Für die iterative Version benutzen wir einen expliziten Stack, der diejenigen Knoten enthält, die wir schon entdeckt haben, aber für die wir noch nicht alle ausgehenden Kanten inspiziert haben. Der Stack speichert zu jedem solchen Knoten noch einen Iterator für die Menge der jeweils inzidenten Kanten. Zu Beginn ist das v`.found` Attribut für alle Knoten `false`.

```
// Initialisiere S
S <- new Stack
// process s
s.found <- true
S.push((s, s.outgoingEdges()))
while not S.isEmpty() do
  (v, e) <- S.top()
  // finde einen unentdeckten Nachbarn
  // des aktuellen Knoten
  do
    if e.hasNext() then
      w <- e.next().opposite(v)
    else
      w <- NULL
  while w != NULL and w.found
  // falls v keinen unentdeckten Nachbarn mehr hat,
  // nimm ihn vom Stack
  if w = NULL then
    S.pop()
  // ansonsten betrachte diesen Nachbarn als naechstes
  else
    // process w
    w.found <- true
    S.push((w, w.outgoingEdges()))
```

Der Stack entspricht dem "Ariadnefaden": die Knoten in `S` stellen einen Pfad von v zurück zu s dar.

4 Breitensuche

Zu Beginn ist v.`found` = `false` für alle Knoten v.

```
Q <- new Queue
s.found <- true
Q.enqueue(s)
while not Q.isEmpty() do
  v <- Q.dequeue()
  //process v
  for e in v.outgoingEdges() do
    w <- e.opposite(v)
    if not w.found then
      w.found <- true
      Q.enqueue(w)
```

5 BFS mit kürzesten Wegen

Breitensuche lässt sich leicht modifizieren, um kürzeste Wege in ungewichteten Graphen zu berechnen. Jetzt speichert jeder Knoten zusätzlich noch einen Wert $v.d$, der den Abstand zwischen v und s angibt, und einen Zeiger v.`pred`, der zu dem Vorgängerknoten auf einem kürzesten Weg von v zu s zeigt.

```
Q <- new Queue
s.found <- true
s.d <- 0       // Neu
s.pred <- NULL // Neu
Q.enqueue(s)
while not Q.isEmpty() do
  v <- Q.dequeue()
  //process v
  for e in v.outgoingEdges() do
    w <- e.opposite(v)
    if not w.found then
      w.found <- true
      w.d <- v.d + 1  // Neu
      w.pred <- v     // Neu
      Q.enqueue(w)
```

6 Dijkstra

Der Algorithmus von Dijkstra verallgemeinert den obigen BFS-Algorithmus auf gewichtete Kanten, wobei alle Längen nichtnegativ sein müssen. Jede Kante hat jetzt eine Länge e.`length` ≥ 0.

Es gibt folgende Unterschiede:

1. Wir verwenden eine Prioritätswarteschlange statt einer einfachen Schlange.

2. Die Interpretation für $v.d$ und v.`pred` ist ein bisschen anders. Für alle Knoten, für die v.`found` = `true` ist und die nicht mehr in `Q` sind, ist $v.d$ der Abstand von s zu v und v.`pred` der Vorgänger auf einem kürzesten Weg. Für alle Knoten, bei denen v.`found` = `true` ist und die noch in `Q` enthalten sind, ist $v.d$ die kürzeste Länge eines Weges von s nach v, der nur Knoten w benutzt, für die (w.`found` = `true` $\wedge\ w \notin$ `Q`) gilt. v.`pred` ist der Vorgänger von v auf einem solchen Weg. Diese Information muss man aktualisieren, wenn man einen Knoten aus `Q` entnimmt.

```
Q <- new PrioQueue
s.found <- true
s.d <- 0
s.pred <- NULL
Q.insert(s, 0)
while not Q.isEmpty() do
  v <- Q.extractMin()
  for e in v.outgoingEdges() do
    w <- e.opposite(v)
    if not w.found then
      w.found <- true
      w.d <- v.d + e.length
      w.pred <- v
      Q.insert(w, w.d)
    else if w is in Q then
      // Neu: hier muessen wir
      // ueberpruefen, ob der Weg
      // ueber v nach w kuerzer ist,
      // als der alte Weg.
      if v.d + e.length < w.d then
        w.d <- v.d + e.length
        w.pred <- v
        Q.decreaseKey(w, w.d)
```

Man kann den Algorithmus vereinfachen, indem man v.`found` = `false` durch $v.d = \infty$ darstellt. Dann kann man sich das v.`found` Attribut sparen und braucht eine `if`-Abfrage weniger.

```
Q <- new PrioQueue
// Fuege alle Knoten in Q ein
// und gibt ihnen Abstand INFTY
for v in vertices() do
  v.d <- INFTY
  v.pred <- NULL
  Q.insert(v, INFTY)
// nur s hat Abstand 0
s.d <- 0
Q.decreaseKey(s, 0)
while not Q.isEmpty() do
  v <- Q.extractMin()
  for e in v.outgoingEdges() do
    w <- e.opposite(v)
    if v.d + e.length < w.d then
      w.d <- v.d + e.length
      w.pred <- v
```

```
Q.decreaseKey(w, w.d)
```