

Graph-Traversal und Algorithmen

Breitensuche



Immer wenn ein Knoten besucht wird, merken wir uns alle seine bislang unbesuchten Nachbarn in einer Queue. Der nächste besuchte Knoten ist der erste in der Queue.

```
1 visitBFS(v: Vertex, g:Graph) {
2
3   // Lege den ersten Knoten in die Queue
4   schlange :Queue<Vertex> = new Queue<Vertex>();
5   schlange.enqueue(v);
6
7   while ( !schlange.isEmpty() ) {
8       // Hole den aktuellen Knoten
9       v = schlange.dequeue();
10
11      // Füge ALLE unbesuchten Nachbarn in die Queue ein
12      Fuer alle Kanten (v,u) mit u unbesucht {
13          schlange.enqueue(u);
14          u.setVisited(true);
15      }
16  }
17 }
```

Immer wenn ein Knoten u bei der Breitensuche vom Knoten v in die Queue getan wird, ist die Distanz von u zum Startknoten gleich $\text{dist}(u) = \text{dist}(v) + 1$.

Man kann somit bei der Breitensuche die Distanzen vom Startknoten berechnen.

Tiefensuche

rekursiv:

```

1 visitDFS(hier : Vertex) {
2     hier.setVisited(true); // Markiere als besucht
3
4     hier.preorderVisit(); // Erstes Betreten
5
6     Fuer alle Kanten (hier,u) {
7
8         // Falls u unbesucht ist ...
9         if ( u.isVisited() != true ) {
10             // ... gehe dort hin}
11             visitDFS(u);
12         }
13
14     hier.inorderVisit(u); // zwischen den Kanten
15
16 }
17
18 hier.postorderVisit(); // Abschlussarbeiten
19 }

```

iterativ mit Stack:

```

1 void visitDFS(hier : Vertex) {
2     var v : Vertex;
3     var u : Vartex;
4
5     // Lege den ersten Knoten auf den Stack
6     var faden : Stack<Vertex> = new Stack<Vertex>();
7     faden.push(hier);
8
9     while ( !faden.isEmpty() ) {
10         // Hole den aktuellen Knoten
11         v = faden.top();
12         if ( Es existiert Kante (v,u) mit u unbesucht ) {
13             // Gehe zu u, wickel den Faden ab
14             faden.push(u);
15             u.setVisited(true);
16             u.preorderVisit();

```

```

17     } else {
18         // Gehe zurück, wickel den Faden auf
19         v.postorderVisit();
20         faden.pop();
21     }
22 }
23 }

```

Jarnik-Prim

Ermittelt den minimalen Spannbaum eines zusammenhängenden gewichteten Graphen.



Finde immer den **nicht angeschlossenen** Knoten, der am **günstigsten** angeschlossen werden kann.

```

for each vertex v {
    connected[v] = false;
    cost[v] = INFTY;
    neighbor[v] = null;
}

// Startknoten s
cost[s] = 0;
int totalCost = 0; // Die Gesamtkosten

Für 2 ... n tue { // Jeder Knoten einmal
    Suche Knoten v mit connected[v] == false und min(cost[v])
    connected[v] = true; // Schließe v an
    totalCost = totalCost + cost[v]; // addiere Kosten
    Für alle Nachbarn u von v mit connected[u] == false tue {
        if ( w(v,u) < cost[u] ) { // Falls er billiger wird
            cost[u] = w(v,u); // senke die Kosten und
            neighbor[u] = v; // merke Dir den neuen Nachbarn.
        }
    }
}

```

```
}  
}
```

Laufzeit:

- Die Vorbereitung benötigt Zeit $\mathcal{O}(n)$.
- Es werden n Runden durchgeführt, in denen die folgenden Schritte gemacht werden:
 - Suche des Knotens mit minimalen Kosten.
 - Anpassen der Kosten für die Nachbarn.
- Die Suche erfordert mit einer linearen Suche Zeit $\mathcal{O}(n)$.
- Die Anpassung erfordert für jede Kante $\mathcal{O}(1)$.

Laufzeit des Algorithmus von Jarnik und Prim

Der Algorithmus benötigt bei Verwendung der linearen Suche $\mathcal{O}(n \cdot n + m) = \mathcal{O}(n^2 + m)$ Zeit.

Kruskal

Ermittelt den minimalen Spannbaum eines zusammenhängenden gewichteten Graphen.



Sortiere die Kanten aufsteigend nach Gewicht. Füge die nächstgünstigste Kante hinzu, die keinen **Kreis** erzeugt.

Struktogramm und Laufzeit:

Der Algorithmus von Kruskal mit Union-Find

Eingabe: Graph $G=(V,E)$ mit Kantengewichten $w(e)$								
$P = \text{new Partition}(V)$	$\mathcal{O}(n)$							
Sortiere alle Kanten nach Gewicht: e_1, \dots, e_m	$\mathcal{O}(m \log m)$							
Initialisiere eine leere Kantenmenge T	$\mathcal{O}(1)$							
Für $i = 1 \dots m$	$m\text{-Mal}$							
Seien u und v die Knoten von e_i	$\mathcal{O}(1)$							
<table border="1"> <tr> <td colspan="2">$P.\text{isSame}(u,v) == \text{false}$</td></tr> <tr> <td>Ja</td><td>Nein</td></tr> <tr> <td>Füge e_i zur Auswahl T hinzu</td><td rowspan="2">\emptyset</td></tr> <tr> <td>$P.\text{union}(u,v)$</td></tr> </table>	$P.\text{isSame}(u,v) == \text{false}$		Ja	Nein	Füge e_i zur Auswahl T hinzu	\emptyset	$P.\text{union}(u,v)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ $\mathcal{O}(n)$
$P.\text{isSame}(u,v) == \text{false}$								
Ja	Nein							
Füge e_i zur Auswahl T hinzu	\emptyset							
$P.\text{union}(u,v)$								

Laufzeit des Kruskal-Algorithmus

Mit der naiven Implementierung von UnionFind hat der Algorithmus von Kruskal eine Laufzeit von $\mathcal{O}(n + m \log m + mn)$ mit der Kantenanzahl m und der Knotenanzahl n .

Dijkstra

Single Source Shortest Path Problem → für einen gegebenen Startknoten kürzeste Wege zu allen anderen Knoten.

Pseudocode:

```

Startknoten in Warteschlange  $W$  aufnehmen
Menge der erledigten Knoten  $E = \emptyset$ 
Kosten des Startknotens mit  $0$  bewerten
Kosten für alle Knoten außer Startknoten mit  $\infty$  bewerten

solange  $W \neq \emptyset$ 

```

wähle Knoten k mit den geringsten Kosten zum Startknoten
 füge k zu W hinzu
 berechne neue Kosten für alle Nachfolger j von k ...
 ... die nicht Element von E sind

falls Kosten zu j über k geringer sind
 aktualisiere Kosten zu j
 aktualisiere Vorgänger von j

füge j zu W hinzu
 entferne k aus W
 füge k zu E hinzu

Iteration		A	B	C	D	E	
0	Kosten	0	∞	∞	∞	∞	
	Vorgänger	-	-	-	-	-	
1	Kosten	0	100	∞	50	∞	
	Vorgänger	-	A	-	A	-	
2	Kosten	0	100	200	50	60	
	Vorgänger	-	A	-	A	D	
3	Kosten	0	100	200	50	60	
	Vorgänger	-	A	B	A	D	
4	Kosten	0	100	200	50	250	
	Vorgänger	-	A	B	A	C	
5	Kosten	0	100	200	50	250	Warteschlange: leer
	Vorgänger	-	A	B	A	C	Erledigt: A, B, C, D, E

Laufzeit:

Laufzeit des Algorithmus von Dijkstra

$$T_{Dijkstra}(n, m) = \mathcal{O}(n) + \sum_{k=1}^n T_{suche}(n, k) + \mathcal{O}(m)$$

Die verbliebenen Knoten werden in einem **Fibonacci-Heap** gespeichert. Dann gilt

$$T_{Dijkstra}(n, m) = \mathcal{O}(n \log(n) + m).$$

Zum Vergleich:

- Lineare Suche: $\mathcal{O}(n^2 + m)$
- Min-Heaps: $\mathcal{O}((n + m) \log(n))$