Wie wird ein Graph dargestellt?

Für einen Graphen $G = (V, E)$, ob gerichtet oder ungerichtet, verwende eine Adjazenzliste A_G:

- $A_G[i]$ zeigt auf eine Liste aller Nachbarn von Knoten i, wenn G ungerichtet ist, oder auf eine Liste aller direkten Nachfolger von Knoten i, wenn G gerichtet ist.

Wie wird ein Graph dargestellt?

Für einen Graphen $G = (V, E)$, ob gerichtet oder ungerichtet, verwende eine Adjazenzliste A_G:

- $A_G[i]$ zeigt auf eine Liste aller Nachbarn von Knoten i, wenn G ungerichtet ist, oder auf eine Liste aller direkten Nachfolger von Knoten i, wenn G gerichtet ist.
- Warum Adjazenzlisten? In vielen Anwendungen wie
  - Tiefen- und Breitensuche,
  - Algorithmen von Dijkstra und Prim

  müssen wir schnell auf Nachbarn oder Nachfolger zugreifen.

Tiefensuche für ungerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum- und Rückwärtskanten zerlegt.

Tiefensuche für ungerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum- und Rückwärtskanten zerlegt.

- Wenn Tiefensuche im Knoten v beginnt, werden alle von v aus erreichbaren Knoten besucht: Die Bäume des Walds entsprechen den Zusammenhangskomponente von G.

Tiefensuche für ungerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum- und Rückwärtskanten zerlegt.

- Wenn Tiefensuche im Knoten v beginnt, werden alle von v aus erreichbaren Knoten besucht: Die Bäume des Walds entsprechen den Zusammenhangskomponente von G.
- Wir erhalten schnelle Algorithmen für
  - die Bestimmung aller Zusammenhangskomponenten,
  - die Überprüfung, ob der Graph ein Wald oder ein Baum ist
  - die Überprüfung, ob G bipartit ist und
  - um aus einem ungerichteten Labyrinth herauszufinden.

Tiefensuche für gerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum-, Rückwärts-, Vorwärts- und „Rechts-nach-Links“ Querkanten zerlegt.

Tiefensuche für gerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum-, Rückwärts-, Vorwärts- und „Rechts-nach-Links“ Querkanten zerlegt.

- Wenn Tiefensuche im Knoten v beginnt, werden alle von v aus erreichbaren Knoten besucht.

Tiefensuche für gerichtete Graphen G:

Tiefensuche erzeugt einen Wald, die Kanten des Graphen werden in Baum-, Rückwärts-, Vorwärts- und „Rechts-nach-Links“ Querkanten zerlegt.

- Wenn Tiefensuche im Knoten v beginnt, werden alle von v aus erreichbaren Knoten besucht.
- Wir erhalten schnelle Algorithmen für
  - die Überprüfung, ob G kreisfrei ist (keine Rückwärtskanten),
  - die Überprüfung, ob G stark zusammenhängend ist, also ob es für jedes Knotenpaar (u, v) einen Weg von u nach v gibt und
  - um aus einem gerichteten Labyrinth herauszufinden.

Breitensuche

- Breitensuche, wenn im Knoten v gestartet, erzeugt einen Baum kürzester Wege von v zu allen anderen Knoten.
  - Die Länge eines Weges ist die Anzahl seiner Kanten.

Breitensuche

- Breitensuche, wenn im Knoten v gestartet, erzeugt einen Baum kürzester Wege von v zu allen anderen Knoten.
  - Die Länge eines Weges ist die Anzahl seiner Kanten.
- Leider versagt Breitensuche, wenn Kanten mit Längen beschriftet werden.
  - Die Länge eines Weges ist die Summe seiner Kantenlängen.

Wir benötigen den Algorithmus von Dijkstra.

Dijkstra's Algorithmus

Für einen Startknoten s berechnet der Algorithmus von Dijkstra einen Baum kürzester Wege.

Dijkstra's Algorithmus

Für einen Startknoten s berechnet der Algorithmus von Dijkstra einen Baum kürzester Wege.

- Diskstra's Algorithmus wird mit einem Heap von Knoten implementiert. Die Priorität eines Knotens v stimmt mit der Länge eines kürzesten, in v endenden S-Weges überein.

Dijkstra's Algorithmus

Für einen Startknoten s berechnet der Algorithmus von Dijkstra einen Baum kürzester Wege.

- Diskstra's Algorithmus wird mit einem Heap von Knoten implementiert. Die Priorität eines Knotens v stimmt mit der Länge eines kürzesten, in v endenden S-Weges überein.
- Die Laufzeit von Dijkstra's Algorithmus für Graphen mit n Knoten und m Kanten ist $O((n+m) \cdot \log_2 n)$: Warum?

Dijkstra's Algorithmus

Für einen Startknoten s berechnet der Algorithmus von Dijkstra einen Baum kürzester Wege.

- Diskstra's Algorithmus wird mit einem Heap von Knoten implementiert. Die Priorität eines Knotens v stimmt mit der Länge eines kürzesten, in v endenden S-Weges überein.
- Die Laufzeit von Dijkstra's Algorithmus für Graphen mit n Knoten und m Kanten ist $O((n+m) \cdot \log_2 n)$: Warum?
- Wie aktualisiert man die Distanzwerte nach Aufnahme eines Knotens w in die Menge S?
  Welche Knoten müssen überhaupt aktualisiert werden?

Dijkstra's Algorithmus

Für einen Startknoten s berechnet der Algorithmus von Dijkstra einen Baum kürzester Wege.

- Diskstra's Algorithmus wird mit einem Heap von Knoten implementiert. Die Priorität eines Knotens v stimmt mit der Länge eines kürzesten, in v endenden S-Weges überein.
- Die Laufzeit von Dijkstra's Algorithmus für Graphen mit n Knoten und m Kanten ist $O((n+m) \cdot \log_2 n)$: Warum?
- Wie aktualisiert man die Distanzwerte nach Aufnahme eines Knotens w in die Menge S?
  Welche Knoten müssen überhaupt aktualisiert werden?

Warum ist Dijkstra's Algorithmus korrekt?
Der Begriff der S-Wege ist fundamental.

Kürzeste kreuzende Kanten

Die Algorithmen von Prim und Kruskal bestimmen minimale Spannbäume indem jeweils kürzeste S-kreuzende Kanten gewählt werden.

- Wie wählt man S in Prim's Algorithmus
- und wie in Kruksal's Algorithmus?

Prim's Algorithmus

- Prim's Algorithmus baut einen minimalen Spannbaum Kante für Kante auf.
- Der Algorithmus benutzt einen Heap von Knoten;
  - die Priorität eines Knotens v stimmt überein mit dem Gewicht einer kürzesten kreuzenden, in v endenden Kante.

Prim's Algorithmus

- Prim's Algorithmus baut einen minimalen Spannbaum Kante für Kante auf.
- Der Algorithmus benutzt einen Heap von Knoten;
  - die Priorität eines Knotens v stimmt überein mit dem Gewicht einer kürzesten kreuzenden, in v endenden Kante.

- Wie zeigt man, dass Prim's Algorithmus korrekt ist?
- Wie schnell ist Prim's Algorithmus und warum?

Kruskal's Algorithmus

- Kruskal's Algorithmus sortiert die Kanten nach aufsteigendem Gewicht
- und beginnt mit einem Wald von Einzelbäumen.

Kruskal's Algorithmus

- Kruskal's Algorithmus sortiert die Kanten nach aufsteigendem Gewicht
- und beginnt mit einem Wald von Einzelbäumen.
- Der Wald wird Kante für Kante zu einem einzigen Spannbaum vereinigt:
  - Eine Kante e wird genau dann eingesetzt, wenn nach Hinzunahme von e kein Kreis geschlossen wird.

Kruskal's Algorithmus

- Kruskal's Algorithmus sortiert die Kanten nach aufsteigendem Gewicht
- und beginnt mit einem Wald von Einzelbäumen.
- Der Wald wird Kante für Kante zu einem einzigen Spannbaum vereinigt:
  - Eine Kante e wird genau dann eingesetzt, wenn nach Hinzunahme von e kein Kreis geschlossen wird.
  - Für die Implementierung wird die Union-Find Datenstruktur verwandt.
    - ★ Wie wird die Union-Find Datenstruktur implementiert?
    - ★ Welche Laufzeit hat eine Find-Operation, bzw. eine Union-Operation höchstens?

Kruskal's Algorithmus

- Kruskal's Algorithmus sortiert die Kanten nach aufsteigendem Gewicht
- und beginnt mit einem Wald von Einzelbäumen.
- Der Wald wird Kante für Kante zu einem einzigen Spannbaum vereinigt:
  - Eine Kante e wird genau dann eingesetzt, wenn nach Hinzunahme von e kein Kreis geschlossen wird.
  - Für die Implementierung wird die Union-Find Datenstruktur verwandt.
    - ★ Wie wird die Union-Find Datenstruktur implementiert?
    - ★ Welche Laufzeit hat eine Find-Operation, bzw. eine Union-Operation höchstens?

Warum funktioniert Kruskal's Algorithmus und wie schnell ist er?