Datenstrukturen und Algorithmen

Übung 10

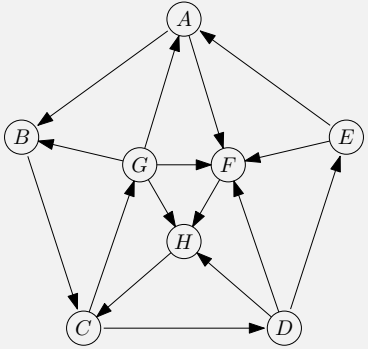
FS 2021

Programm von heute

1. Feedback letzte Übungen
2. Kürzeste Wege
   - Dijkstra
   - Heaps, DecreaseKey und Lazy Deletion
   - Laufzeiten der Algorithmen
   - Dijkstra und negative Kantengewichte?
3. Programmieraufgabe
4. In-Class-Exercise (theoretisch)

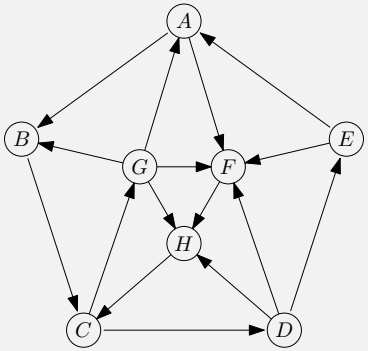
1. Feedback letzte Übungen

Tiefen- und Breitensuche



Start bei A
DFS: A, B, C, D, E, F, H, G
BFS: A, B, F, C, H, D, G, E

Tiefen- und Breitensuche



Start bei A

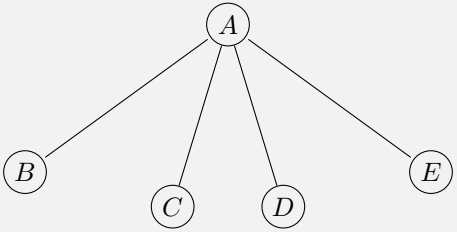
DFS: A, B, C, D, E, F, H, G

BFS: A, B, F, C, H, D, G, E

Es gibt keinen Startknoten, sodass die DFS-Ordnung der BFS-Ordnung entspricht.

Tiefen- und Breitensuche

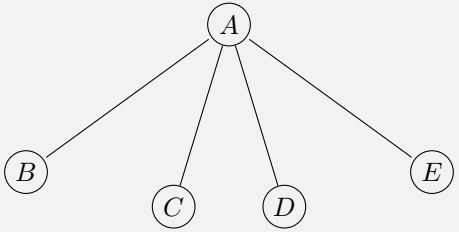
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

Tiefen- und Breitensuche

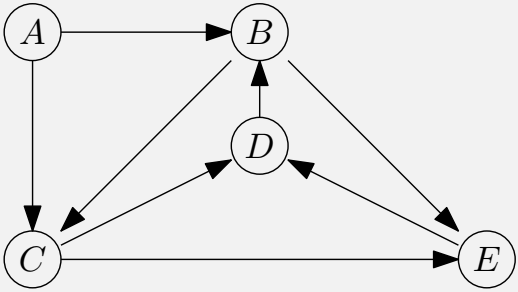
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

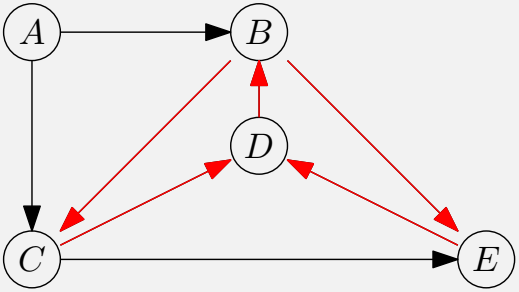
Start bei C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topologische Sortierung



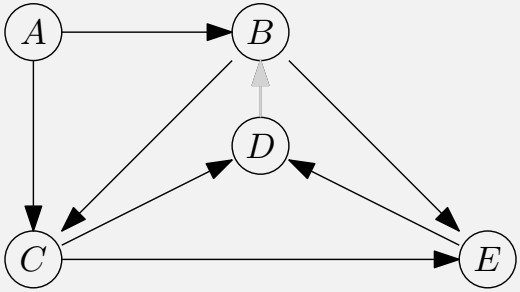
- der Graph hat Kreise

Topologische Sortierung



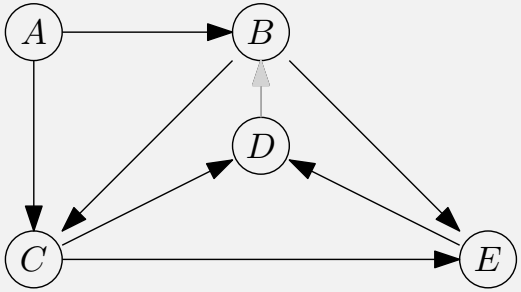
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante

Topologische Sortierung



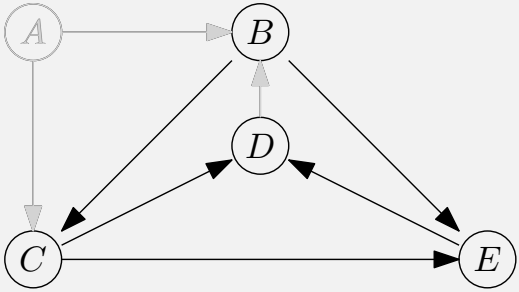
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei

Topologische Sortierung



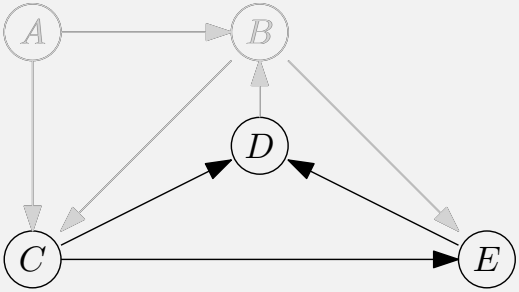
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



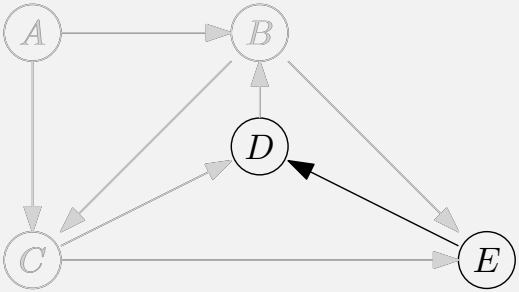
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



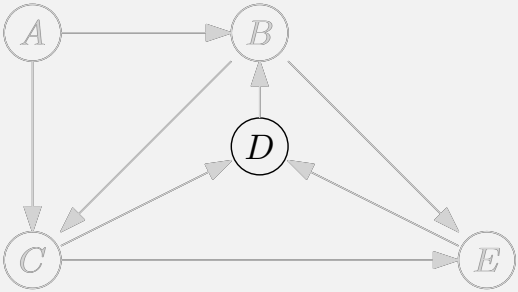
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



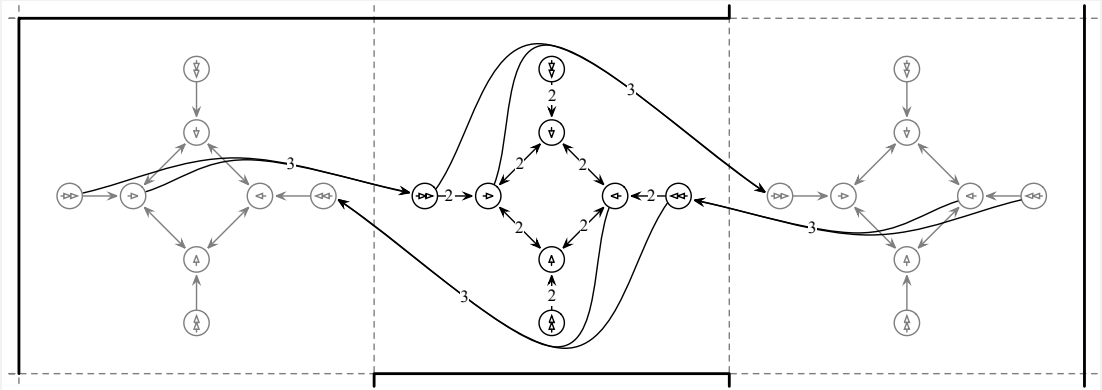
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Aufgabe : Labyrinth

- Roboter muss anhalten um Richtung zu ändern
- Als shortest-path Problem auffassen

Aufgabe : Labyrinth

- Position $\times$ Richtung $\times$ Geschwindigkeit



- Laufzeit?

Aufgabe Labyrinth

- Sei n Anzahl Quadrate. Graph hat $|V| = 8n$ Knoten
- Graph hat $|E| \leq 20n$ Kanten
- Daher hat Dijkstra $\mathcal{O}(|E| + |V| \log |V|)$ Laufzeit $\mathcal{O}(n \log n)$

2. Kürzeste Wege

Allgemeiner Algorithmus

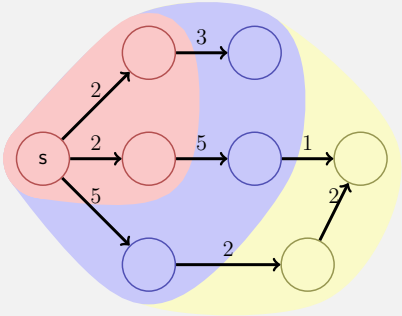
1. Initialisiere d_s und π_s: $d_s[v] = \infty$, $\pi_s[v] =$ null für alle $v \in V$
2. Setze $d_s[s] \leftarrow 0$
3. Wähle eine Kante $(u, v) \in E$

   Relaxiere (u, v):
   if $d_s[v] > d[u] + c(u, v)$ then
   $d_s[v] \leftarrow d_s[u] + c(u, v)$
   $\pi_s[v] \leftarrow u$
4. Wiederhole 3 bis nichts mehr relaxiert werden kann.
   (bis $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Dijkstra (positive Kantengewichte)

Menge V aller Knoten wird unterteilt in

- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.



Algorithmus Dijkstra(G, s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$

Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

foreach $u \in V$ **do**
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow null$
$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ **do**
    $u \leftarrow$ ExtractMin(R)
    foreach $v \in N^+(u)$ **do**
        if $d_s[u] + c(u, v) < d_s[v]$ **then**
            $d_s[v] \leftarrow d_s[u] + c(u, v)$
            $\pi_s[v] \leftarrow u$
            $R \leftarrow R \cup \{v\}$

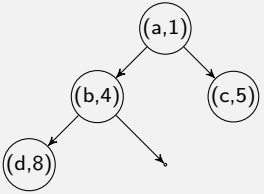
Zur Implementation: Datenstruktur für R?

Relaxieren bei Dijkstra:

if $d_s[u] + c(u, v) < d_s[v]$ **then**
- $d_s[v] \leftarrow d_s[u] + c(u, v)$
- $\pi_s[v] \leftarrow u$
- **if** $v \notin R$ **then**
  - Add(R, v) // Einfügen eines neuen $(v, d(v))$ im Heap zu R
- **else**
  - DecreaseKey(R, v) // Update eines $(v, d(v))$ im Heap zu R

DecreaseKey ?

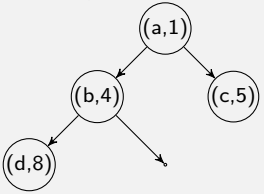
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



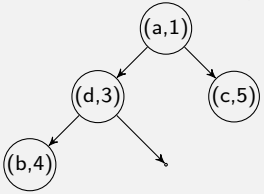
nach DecreaseKey$(d,3)$:

DecreaseKey ?

Heap ($(a,1),(b,4),(c,5),(d,8)$) =



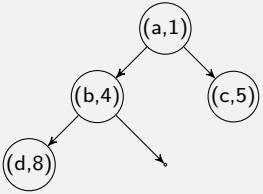
nach DecreaseKey($d,3$):



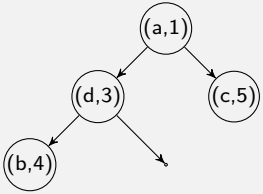
2 Probleme:

DecreaseKey ?

Heap ($(a,1),(b,4),(c,5),(d,8)$) =



nach DecreaseKey($d,3$):

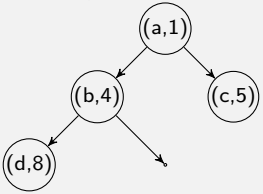


2 Probleme:

- Position von d zuerst nicht bekannt. Suche: $\Theta(n)$
- Position der Knoten kann bei DecreaseKey ändern

Lazy Deletion !

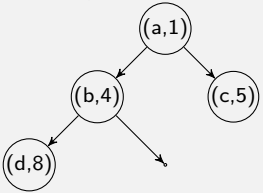
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



Insert$(d,3)$:

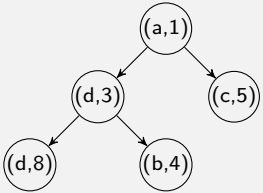
Lazy Deletion !

Heap ($(a,1),(b,4),(c,5),(d,8)$) =



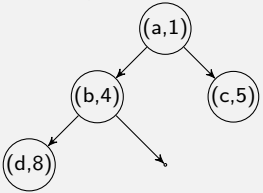
ExtractMin()

Insert$(d,3)$:

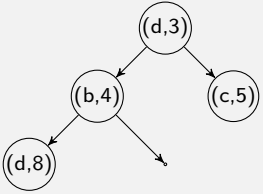


Lazy Deletion !

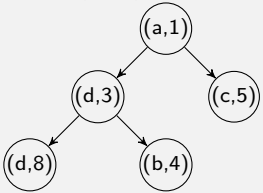
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



ExtractMin() $\rightarrow (a,1)$



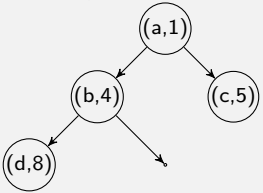
Insert$(d,3)$:



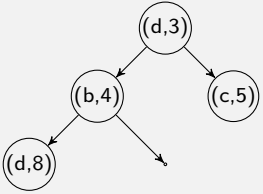
ExtractMin()

Lazy Deletion !

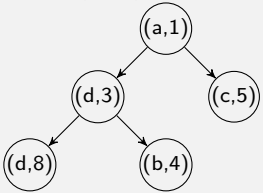
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



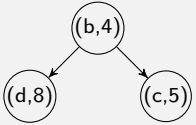
ExtractMin() $\rightarrow (a,1)$



Insert$(d,3)$:



ExtractMin() $\rightarrow (d,3)$



Späteres ExtractMin() $\rightarrow (d,8)$ muss ignoriert werden

Laufzeit Dijkstra

$n := |V|$, $m := |E|$

- $n\times$ ExtractMin: $\mathcal{O}(n \log n)$
- $m\times$ Insert oder DecreaseKey: $\mathcal{O}(m \log |V|)$
- $1\times$ Init: $\mathcal{O}(n)$
- Insgesamt: $\mathcal{O}((n+m) \log n)$. Für zusammenhängende Graphen: $\mathcal{O}(m \log n)$

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein	Bellman-Ford	$\mathcal{O}(m \cdot n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

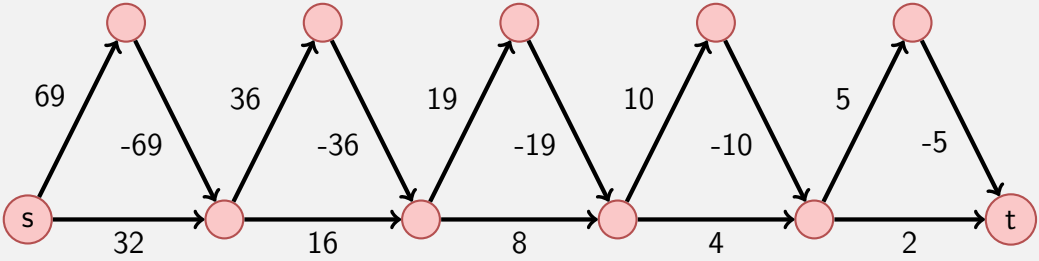
Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

Ein Interessanter Graph



Funktioniert Dijkstra?

Antwort

Antwort

Dijkstra (so wie wir es präsentiert haben) funktioniert auch für Graphen mit negativen Kantengewichten, solange keine negativen Zyklen vorhanden sind. Dijkstra kann dann aber exponentielle Laufzeit haben.

Allgemeine Bewertete Graphen

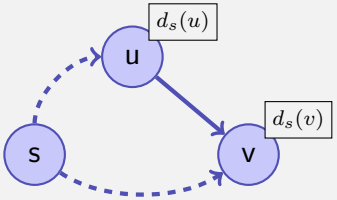
Relax(u, v) $(u, v \in V,\ (u, v) \in E)$

if $d_s(v) > d_s(u) + c(u, v)$ **then**

  $d_s(v) \leftarrow d_s(u) + c(u, v)$

  return true

return false



Problem: Zyklen mit negativen Gewichten können Weg verkürzen: es muss keinen kürzesten Weg mehr geben

Dynamic-Programming-Ansatz (Bellman)

Induktion über Anzahl Kanten. $d_s[i, v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

$$d_s[i, v] = \min\{d_s[i-1, v], \min_{(u,v)\in E}(d_s[i-1, u] + c(u, v))$$

$$d_s[0, s] = 0, d_s[0, v] = \infty \ \forall v \neq s.$$

Algorithmus Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$

Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

foreach $u \in V$ **do**
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ **to** $|V|$ **do**
    $f \leftarrow$ false
    foreach $(u, v) \in E$ **do**
        $f \leftarrow f \vee \mathrm{Relax}(u, v)$
    if $f =$ false **then return** true
return false;

A*-Algorithmus($G, s, t, \hat{h}$)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$, Endpunkt $t \in V$, Schätzung $\widehat{h}(v) \leq \delta(v, t)$

Output: Existenz und Wert eines kürzesten Pfades von s nach t

foreach $u \in V$ **do**
    $d[u] \leftarrow \infty$; $\widehat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null

$d[s] \leftarrow 0$; $\widehat{f}[s] \leftarrow \widehat{h}(s)$; $R \leftarrow \{s\}$; $M \leftarrow \{\}$
while $R \neq \emptyset$ **do**
    $u \leftarrow \text{ExtractMin}_{\widehat{f}}(R)$; $M \leftarrow M \cup \{u\}$
    if $u = t$ **then return** success
    foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**
        $d[v] \leftarrow d[u] + c(u, v)$; $\widehat{f}[v] \leftarrow d[v] + \widehat{h}(v)$; $\pi[v] \leftarrow u$
        $R \leftarrow R \cup \{v\}$; $M \leftarrow M - \{v\}$
return failure

DP-Algorithmus Floyd-Warshall(G)

Input: Azyklischer Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade d

$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Laufzeit: $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

Algorithmus Johnson(G)

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade D.

Neuer Knoten s. Berechne $G' = (V', E', c')$
if BellmanFord(G', s) = false **then** return "graph has negative cycles"
foreach $v \in V'$ **do**
    $h(v) \leftarrow d(s, v)$ // d aus BellmanFord Algorithmus
foreach $(u, v) \in E'$ **do**
    $\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
foreach $u \in V$ **do**
    $\tilde{d}(u, \cdot) \leftarrow$ Dijkstra$(\tilde{G}', u)$
    foreach $v \in V$ **do**
        $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Vergleich der Verfahren

Algorithmus			Laufzeit
Dijkstra (Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert \log \lvert V\rvert)$
Dijkstra (Fibonacci-Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$ *
Bellman-Ford		1:n	$\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$
Floyd-Warshall		n:n	$\Theta(\lvert V\rvert^3)$
Johnson		n:n	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert \cdot \log \lvert V\rvert)$
Johnson (Fibonacci-Heap)		n:n	$\mathcal{O}(\lvert V\rvert^2 \log \lvert V\rvert + \lvert V\rvert \cdot \lvert E\rvert)$ *

* amortisiert

Johnson ist besser als Floyd-Warshall für dünn besetzte Graphen ($|E| \approx \Theta(|V|)$).

3. Programmieraufgabe

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

- Intuitiv: Wenn viele verbundene Knoten nahe bei v liegen, dann ist $C(v)$ klein.
- „Wie zentral ist ein Knoten in seiner Zusammenhangskomponente?“

Alle kürzesten Pfade

- Wir brauchen $d(u, v)$ für alle Knotenpaare (u, v).
- $\Longrightarrow$ berechne alle kürzesten Pfade mit Floyd-Warshall. (`APSH.h`)

```
template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m)
{
  // your code here
}
```

- Das Feld `m` soll mit den Distanzen überschrieben werden.
- Achtung: anfangs bedeutet `0` „keine Kante“.
- Ungerichteter Graph: `m[i][j] == m[j][i]`

Closeness Centrality

Centrality.h

```
void printCentrality(unsigned n, vector<vector<unsigned>>
        adjacencies, vector<string> names)
{
  for(unsigned i = 0; i < n; ++i)
  {
    cout << names[i] << ": ";
    unsigned centrality = 0;
    // TODO: compute centrality of vertex i here
    cout << centrality << endl;
  }
}
```

Closeness Centrality: Eingabedaten

- Der Eingabegraph beschreibt die Zusammenarbeit von gewissen Autoren an wissenschaftlichen Publikationen.
- Die Knoten des Graphen stehen für die Co-Autoren des Mathematikers Paul Erdős.
- Wenn sie zusammen eine Arbeit veröffentlicht haben, sind sie durch eine Kante verbunden.
- Quelle: `https://oakland.edu/enp/thedata/`

Closeness Centrality: Output

```
vertices: 511
ABBOTT, HARVEY LESLIE                : 1625
ACZEL, JANOS D.                      : 1681
AGOH, TAKASHI                        : 2132
AHARONI, RON                         : 1578
AIGNER, MARTIN S.                    : 1589
AJTAI, MIKLOS                        : 1492
ALAOGLU, LEONIDAS*                   : 0
ALAVI, YOUSEF                        : 1561
...
```

Wo kommt die 0 her?

Dijkstra and A*

Edge Datenstruktur

- Speichert nur Länge und Ziel
- Start Node durch `get_adj(src)` oder `std::map<NodeP,Edge> path` gegeben

Dijkstra and A*

Edge Datenstruktur

- Speichert nur Länge und Ziel
- Start Node durch `get_adj(src)` oder `std::map<NodeP,Edge> path` gegeben

Graph Datenstruktur

- `NodeP`: Shared Pointer auf eine Node
- `std::vector<Edge> get_adj(NodeP src)`: Gibt einen Vektor mit Edges zurück, welche von `src` ausgehen.

Dijkstra and A*

Edge Datenstruktur

- Speichert nur Länge und Ziel
- Start Node durch `get_adj(src)` oder `std::map<NodeP,Edge> path` gegeben

Graph Datenstruktur

- `NodeP`: Shared Pointer auf eine Node
- `std::vector<Edge> get_adj(NodeP src)`: Gibt einen Vektor mit Edges zurück, welche von `src` ausgehen.

`std::map<NodeP,Edge>`

- Bildet einen NodeP auf einen Edge ab.
- `m[u]` gibt einen Edge zurück.

Dijkstra and A*

Node Struct

- Speichert die x und y Koordinaten

Dijkstra and A*

Node Struct

- Speichert die x und y Koordinaten

Manhattan Distance:

- $d = |\Delta x| + |\Delta y|$

Dijkstra and A*

Node Struct

- Speichert die x und y Koordinaten

Manhattan Distance:

- $d = |\Delta x| + |\Delta y|$

`std::pair`

- Zugriff mit `p.first` und `p.second`

4. In-Class-Exercise (theoretisch)

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

Tipp: G ist kreisfrei, Sie können also zuerst topologisch sortieren.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

Vorgänger merken!

Fragen?