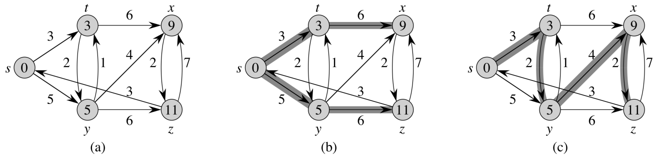
Korteste veje

Korteste veje i vægtede grafer

Længde af sti = sum af vægte af kanter på sti.

$\delta(u, v)$ = længden af en korteste sti fra u til v. Sættes til ∞ hvis ingen sti findes.

Single-source shortest-path problemet: Givet $s \in V$, find $\delta(s, v)$ (og en konkret sti af denne længde) for alle $v \in V$.

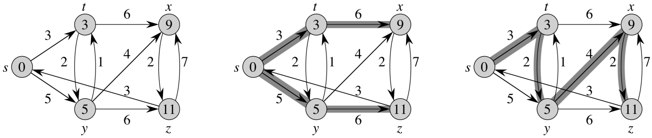


Korteste veje i vægtede grafer

Længde af sti = sum af vægte af kanter på sti.

$\delta(u, v)$ = længden af en korteste sti fra u til v. Sættes til ∞ hvis ingen sti findes.

Single-source shortest-path problemet: Givet $s \in V$, find $\delta(s, v)$ (og en konkret sti af denne længde) for alle $v \in V$.

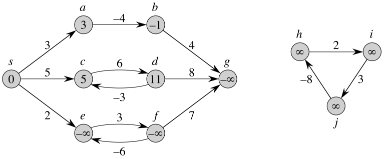


(a) (b) (c)

Bemærk at prefixer af korteste veje selv må være korteste veje: hvis $v_1, v_2, \ldots, v_k$ er en korteste vej fra v_1 til v_k, så er $v_1, v_2, \ldots, v_i$ en korteste vej fra v_1 til v_i for alle $i \leq k$ (ellers kan vejen fra v_1 til v_k gøres kortere).

Korteste veje i vægtede grafer

Problemet er ikke veldefineret, hvis der findes kredse (som kan nås fra s) med negativ sum, idet der så findes veje med vilkårlig lav længde:



Omvendt: hvis der ikke findes sådanne negative kredse, kan vi nøjes med at se på simple stier (ingen gentagelser af knuder på stien). Der er et endeligt antal sådanne stier (højst $n!$), så "længde af korteste sti" er veldefineret.

Relaxation – en generel teknik til at finde korteste veje

Idé: Brug kanter til at udbrede information fra knude til knude om længder af kendte stier. Kaldes RELAX af kanten.

Hvis u har information om, at der er en sti fra s til u af længde $u.d$, og (u, v) er en kant af med vægt w, så findes der en sti af længde $u.d + w$ til v. Er det bedre information for v, end hvad den har lige nu?

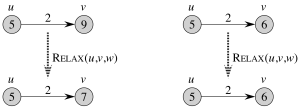
Relaxation – en generel teknik til at finde korteste veje

Idé: Brug kanter til at udbrede information fra knude til knude om længder af kendte stier. Kaldes RELAX af kanten.

Hvis u har information om, at der er en sti fra s til u af længde $u.d$, og (u, v) er en kant af med vægt w, så findes der en sti af længde $u.d + w$ til v. Er det bedre information for v, end hvad den har lige nu?

INIT-SINGLE-SOURCE(G, s)
  for each $v \in G.V$
      $v.d = \infty$
      $v.\pi = \text{NIL}$
  $s.d = 0$

RELAX(u, v, w)
  if $v.d > u.d + w(u, v)$
      $v.d = u.d + w(u, v)$
      $v.\pi = u$



(a) (b)

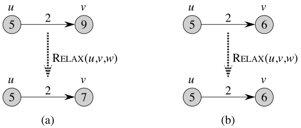
Relaxation – en generel teknik til at finde korteste veje

Idé: Brug kanter til at udbrede information fra knude til knude om længder af kendte stier. Kaldes RELAX af kanten.

Hvis u har information om, at der er en sti fra s til u af længde $u.d$, og (u, v) er en kant af med vægt w, så findes der en sti af længde $u.d + w$ til v. Er det bedre information for v, end hvad den har lige nu?

```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



[Detalje: Implementation af "∞", skal fungere matematisk korrekt med "$>$" og $+$" (bare at bruge `Integer.MAX_VALUE` som "∞" er ikke nok).]

Relaxation

```
INIT-SINGLE-SOURCE(G, s)
 for each v ∈ G.V
      v.d = ∞
      v.π = NIL
 s.d = 0
```

```
RELAX(u, v, w)
 if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```

Vi vil møde en række algoritmer, som starter med INIT-SINGLE-SOURCE og derefter kun ændrer $v.d$ og $v.\pi$ via RELAX.

For sådanne algoritmer gælder følgende invariant (ses nemt ved induktion på antal RELAX):

> Hvis $v.d < \infty$ findes en sti fra s til v af længde $v.d$.

Relaxation

$$\boxed{\text{Hvis } v.d < \infty \text{ findes en sti fra } s \text{ til } v \text{ af længde } v.d.}$$

Relaxation

$$\boxed{\text{Hvis } v.d < \infty \text{ findes en sti fra } s \text{ til } v \text{ af længde } v.d.}$$

Derfor gælder altid $\delta(s,v) \leq v.d$. Argument: hvis $v.d < \infty$ følger det af ovenstående, hvis $v.d = \infty$ gælder $\delta(s,v) \leq v.d$ uanset værdien af $\delta(s,v)$.

Relaxation

$$\boxed{\text{Hvis } v.d < \infty \text{ findes en sti fra } s \text{ til } v \text{ af længde } v.d.}$$

Derfor gælder altid $\delta(s, v) \leq v.d$. Argument: hvis $v.d < \infty$ følger det af ovenstående, hvis $v.d = \infty$ gælder $\delta(s, v) \leq v.d$ uanset værdien af $\delta(s, v)$.

Da $v.d$ kun kan falde ved brug af $\textsc{Relax}$, følger det, at hvis på et tidspunkt $\delta(s, v) = v.d$, vil $v.d$ ikke kunne ændres senere (og dermed kan heller ikke $v.\pi$ ændres, da $v.d$ og $v.\pi$ ændres på samme tidspunkt i koden).

Specielt gælder, at hvis $\delta(s, v) = v.d$ for alle knuder, vil ingen kant (u, v) kunne relaxeres (dvs. $v.d \leq u.d + w(u, v)$ gælder for alle kanter (u, v)).

De korteste stier kan findes via $v.\pi$-pointers

Invariant:

- Mængden S af knuder v med $\delta(s, v) = v.d < \infty$ udgør et træ med $v.\pi$ som parent pointers og s som rod.
- For en knude v i træet vil stien mod roden svare til et baglæns gennemløb af en sti i grafen fra s til v af længde $\delta(s, v)$.

De korteste stier kan findes via $v.\pi$-pointers

Invariant:

- Mængden S af knuder v med $\delta(s,v) = v.d < \infty$ udgør et træ med $v.\pi$ som parent pointers og s som rod.
- For en knude v i træet vil stien mod roden svare til et baglæns gennemløb af en sti i grafen fra s til v af længde $\delta(s,v)$.

Dette vises ved induktion på antal RELAX.

Basis: Lige efter initialisering er s den eneste knude v, som har $v.d < \infty$. Da $s.d = 0$ og $s.\pi = \text{NIL}$ efter initialisering, er $S = \{s\}$ og invarianten opfyldt med et træ af størrelse én, hvis blot $\delta(s,s) = 0$.

[Bemærk at $\delta(s,s) \neq 0$ kun er muligt hvis s ligger på en negativ kreds. Men så gælder for alle knuder v enten $\delta(s,v) = -\infty$ (hvis v kan nås fra s) eller $\delta(s,v) = \infty$ (hvis v ikke kan nås fra s). Hvis $v.d < \infty$, svarer $v.d$ til længden af en konkret sti (se tidligere), og er derfor forskellig fra $-\infty$. Så S er altid tom, og der er intet at vise.]

De korteste stier kan findes via $v.\pi$-pointers

Induktionsskridt: For en $\textsc{Relax}$ som ikke ændrer noget, er der intet at vise. For en $\textsc{Relax}$, som ændrer $v.d$ (og dermed $v.\pi$): her kan v ikke være i S før $\textsc{Relax}$ (da $\delta(s, v) < v.d$ på det tidspunkt), så alle eksisterende knuder i træet er uændrede (inkl. deres parent pointers). Hvis v indlemmes i S pga. denne $\textsc{Relax}$, så lad (u, v) være kanten, som blev relaxeret, og lad w være dens vægt. Vi har så $\delta(s, v) = v.d = u.d + w$. Derfor må $\delta(s, u) = u.d$ gælde (hvis der var en vej kortere end $u.d$ til u, var der en vej kortere end $u.d + w$ til v) og $u.d < \infty$ gælder også (ellers ville $\textsc{Relax}$ ikke ske). Så u er med i S, får v som barn, og sætningen gælder klart igen.

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid:

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: n INSERT (eller én BUILD-HEAP), n EXTRACT-MIN og m DECREASE-KEY (i RELAX).

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: n INSERT (eller én BUILD-HEAP), n EXTRACT-MIN og m DECREASE-KEY (i RELAX). I alt $O(m \log n)$ hvis prioritetskøen implementeres med en heap.

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

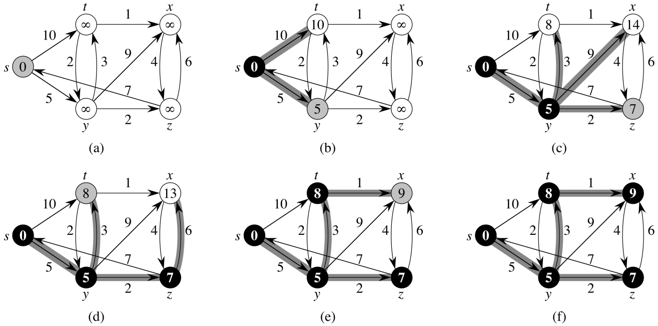
```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: n INSERT (eller én BUILD-HEAP), n EXTRACT-MIN og m DECREASE-KEY (i RELAX). I alt $O(m \log n)$ hvis prioritetskøen implementeres med en heap.

Invariant: Når u indlemmes i S (dvs. udtages med en EXTRACT-MIN) er $u.d = \delta(s, u)$ (hvis alle kantvægte er ≥ 0).

Bevis for invariant: Et induktionsbevis (gennemgået på tavle). Af invarianten følger at algoritmen er korrekt (da alle knuder er i S til sidst).

Dijkstra, eksempel



Path-relaxation lemma

Dijkstra antager ikke-negative vægte. Vi kigger nu på algoritmer, som kan klare negative vægte (men naturligvis ikke negative kredse, som gør korteste vej problemet udefineret).

Path-relaxation lemma

Dijkstra antager ikke-negative vægte. Vi kigger nu på algoritmer, som kan klare negative vægte (men naturligvis ikke negative kredse, som gør korteste vej problemet udefineret).

Vi starter med flg. lemma:

Lemma: Hvis $s = v_1, v_2, \ldots, v_k = v$ er en korteste vej fra s til v, og en algoritme laver $\textsc{Relax}$ på kanterne $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ efter tur (med en vilkårlig mængde $\textsc{Relax}$ af andre kanter mellem disse $\textsc{Relax}$), da er $\delta(s, v) = v.d$ efter den sidste af disse $\textsc{Relax}$.

Bevis: Det ses ved induktion på i, at efter der er lavet $\textsc{Relax}$ på kanten (v_{i-1}, v_i) i sekvensen ovenfor, kan $v_i.d$ højst være lig summen af vægtene af de første $i - 1$ kanter i stien.

Så efter den sidste af disse $\textsc{Relax}$ er $v.d \leq \delta(s, v)$, eftersom stien er en korteste sti til v. Da $\delta(s, v) \leq v.d$ altid gælder, er $\delta(s, v) = v.d$.

Algoritme for DAGs [unknown]

Recall: DAG $=$ Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
    for each vertex $v \in G.Adj[u]$
      RELAX(u, v, w)

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
    for each vertex $v \in G.Adj[u]$
      RELAX(u, v, w)

Køretid:

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

```
DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: $O(n+m)$.

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

```
DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: $O(n+m)$.

Sætning: Når algoritmen stopper er $v.d = \delta(s, v)$ for alle $v \in V$.

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

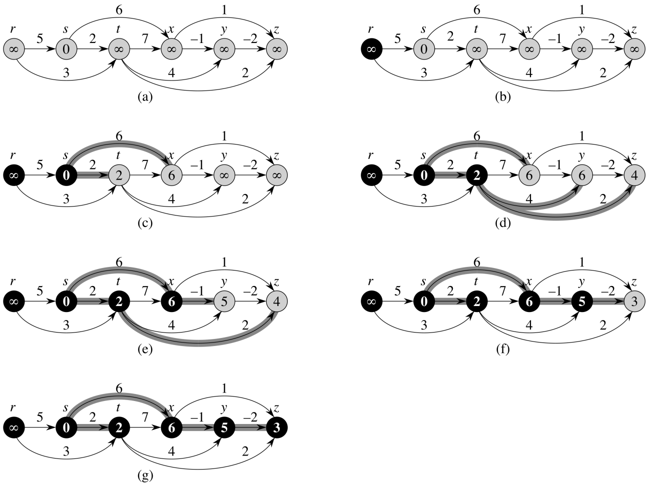
DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
    for each vertex $v \in G.Adj[u]$
      RELAX(u, v, w)

Køretid: $O(n+m)$.

Sætning: Når algoritmen stopper er $v.d = \delta(s, v)$ for alle $v \in V$.

Bevis: For en knude v med en sti fra s til v: alle knuder på en korteste sti er blevet relaxeret i rækkefølge (hvorved korrekte δ-værdier sættes på denne sti pga. path-relaxation lemma). For alle andre knuder gælder $\infty = \delta(s, v)$ så korrekthed her følger af $\delta(s, v) \leq v.d$.

Algoritmen for DAG, eksempel



Bellman-Ford-Moore [1956-57-58]

```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| − 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

Bellman-Ford-Moore [1956-57-58]

```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| − 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

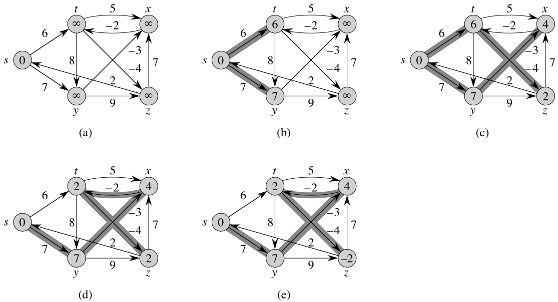
Køretid:

Bellman-Ford-Moore [1956-57-58]

```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| - 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

Køretid: $O(nm)$

Bellman-Ford-Moore [1956-57-58]



[Relaxeringsrækkefølge:
$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.]

Bellman-Ford, korrekthed

Idéen bag Bellman-Ford er, at den vedligeholder følgende invariant:

Efter i iterationer af første **for**-løkke gælder $v.d = \delta(s, v)$ for alle knuder v, der har en korteste vej med højst i kanter.

Denne invariant følger direkte af path-relaxation lemmaet.

Bellman-Ford, korrekthed

Idéen bag Bellman-Ford er, at den vedligeholder følgende invariant:

> Efter i iterationer af første **for**-løkke gælder $v.d = \delta(s, v)$ for alle knuder v, der har en korteste vej med højst i kanter.

Denne invariant følger direkte af path-relaxation lemmaet.

Mere præcist gælder følgende for Bellman-Ford:

Sætning: Hvis der findes en negativ kreds, som kan nås fra s, svarer Bellman-Ford FALSE. Ellers svarer den TRUE, og $v.d = \delta(s, v)$ for alle $v \in V$ når den stopper.

Bellman-Ford, korrekthed

Bevis:

Case 1: der er ingen negative kredse, som kan nås fra s.

Så har alle knuder, som kan nås fra s, en simpel korteste vej (en vej uden gentagelser af knuder). En sådan vej har højst n knuder, og derfor højst $n-1$ kanter. Af invarianten ovenfor gælder $v.d = \delta(s, v)$ for disse knuder, når den første **for**-løkke slutter. For knuder, der ikke kan nås fra s, gælder dette allerede efter initialiseringen i starten. Så $v.d = \delta(s, v)$ for alle knuder, når den første **for**-løkke slutter.

Bellman-Ford, korrekthed

Bevis:

Case 1: der er ingen negative kredse, som kan nås fra s.

Så har alle knuder, som kan nås fra s, en simpel korteste vej (en vej uden gentagelser af knuder). En sådan vej har højst n knuder, og derfor højst $n-1$ kanter. Af invarianten ovenfor gælder $v.d = \delta(s, v)$ for disse knuder, når den første **for**-løkke slutter. For knuder, der ikke kan nås fra s, gælder dette allerede efter initialiseringen i starten. Så $v.d = \delta(s, v)$ for alle knuder, når den første **for**-løkke slutter.

Når $v.d = \delta(s, v)$ for alle knuder, kan $\textsc{Relax}$ ikke ændre nogen $v.d$ længere (jvf. tidligere observation). Derfor svarer bliver **if**-casen i den anden **for**-løkke aldrig sand, og algoritmen svarer TRUE.

Bellman-Ford, korrekthed

Case 2: der er en negativ kreds C, som kan nås fra s.

Vi bemærker først, at hvis en knude v kan nås fra s, kan den også nås via en simpel sti (en sti uden gentagelser blandt knuder). En sådan sti har højst n knuder.

Det er let at se via induktion over i, at efter i iterationer af første **for**-løkke er d-værdien endelig for de første $i + 1$ knuder på denne simple sti. Derfor gælder $v.d < \infty$ ved **for**-løkkens afslutning.

Specielt gælder dette alle knuder på C (de kan alle nås fra s).

Bellman-Ford, korrekthed

Antag, at Bellman-Ford i Case 2 ikke svarer FALSE. Så gælder ved algoritmens afslutning

$$v_{i+1}.d \leq v_i.d + w(v_i, v_{i+1})$$

for $1 \leq i \leq k$ (med $v_{k+1} = v_1$). Og dermed gælder

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_i.d + \sum_{i=1}^{k} w(v_i, v_{i+1}).$$

Da $v_i.d < \infty$ for alle i, er de to første summer ikke bare ens, men også $< \infty$, så de kan trækkes fra og give

$$0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1}),$$

i modstrid med at kredsen er negativ. Så algoritmen må svare FALSE. $\square$

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte):

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Eller: køre Bellman-Ford-Moore fra hver source $s \in V$ (hvis der er negative vægte):

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Eller: køre Bellman-Ford-Moore fra hver source $s \in V$ (hvis der er negative vægte): $O(n^2 m)$ tid.

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Eller: køre Bellman-Ford-Moore fra hver source $s \in V$ (hvis der er negative vægte): $O(n^2 m)$ tid.

En anden mulighed: Floyd-Warshalls algoritme. $O(n^3)$ tid. Klarer negative vægte.

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Eller: køre Bellman-Ford-Moore fra hver source $s \in V$ (hvis der er negative vægte): $O(n^2 m)$ tid.

En anden mulighed: Floyd-Warshalls algoritme. $O(n^3)$ tid. Klarer negative vægte.

Endnu en mulighed: Johnsons algoritme. Kører i $O(nm \log n)$ tid. Klarer negative vægte.

Floyd-Warshalls algoritme [1962]

Bruger ikke RELAX, er i stedet baseret på dynamisk programmering.

Floyd-Warshalls algoritme [1962]

Bruger ikke RELAX, er i stedet baseret på dynamisk programmering.

Input er grafen i adjacency-matrix repræsentationen i en variant W med vægte på kanter: $w_{ii} = 0$, $w_{ij} = w(i, j)$ hvis $(i, j) \in E$, $w_{ij} = \infty$ ellers.

Floyd-Warshalls algoritme [1962]

Bruger ikke RELAX, er i stedet baseret på dynamisk programmering.

Input er grafen i adjacency-matrix repræsentationen i en variant W med vægte på kanter: $w_{ii} = 0$, $w_{ij} = w(i, j)$ hvis $(i, j) \in E$, $w_{ij} = \infty$ ellers.

Output er også på matrice-form:

$D = (d_{ij})$, $d_{ij} = \delta(v_i, v_j) =$ længden af en korteste sti fra v_i til v_j. Sættes til ∞ hvis ingen sti findes.

Floyd-Warshalls algoritme [1962]

Bruger ikke RELAX, er i stedet baseret på dynamisk programmering.

Input er grafen i adjacency-matrix repræsentationen i en variant W med vægte på kanter: $w_{ii} = 0$, $w_{ij} = w(i,j)$ hvis $(i,j) \in E$, $w_{ij} = \infty$ ellers.

Output er også på matrice-form:

$D = (d_{ij})$, $d_{ij} = \delta(v_i, v_j) =$ længden af en korteste sti fra v_i til v_j. Sættes til ∞ hvis ingen sti findes.

$\Pi = (\pi_{ij})$, $\pi_{ij} =$ sidste knude før v_j på en korteste sti fra knude v_i til knude v_j. Sættes til NIL hvis ingen sti findes.

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

FLOYD-WARSHALL(W, n)
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Køretid:

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

Køretid: $O(n^3)$.

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

```
FLOYD-WARSHALL(W, n)
  D^(0) = W
  for k = 1 to n
      let D^(k) = (d_ij^(k)) be a new n × n matrix
      for i = 1 to n
          for j = 1 to n
              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
  return D^(n)
```

Køretid: $O(n^3)$. Plads: $O(n^2)$ (kun forrige $D^{(k)}$ matrice behøves gemmes).

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

FLOYD-WARSHALL(W, n)
  $D^{(0)} = W$
  for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
      for $j = 1$ **to** n
        $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
  return $D^{(n)}$

Køretid: $O(n^3)$. Plads: $O(n^2)$ (kun forrige $D^{(k)}$ matrice behøves gemmes).

Sætning: Når algoritmen stopper er d_{ij} og π_{ij} i den sidste matrice sat korrekt for alle $v_i, v_j \in V$ (hvis ingen negativ kreds er i grafen).

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

FLOYD-WARSHALL(W, n)
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Køretid: $O(n^3)$. Plads: $O(n^2)$ (kun forrige $D^{(k)}$ matrice behøves gemmes).

Sætning: Når algoritmen stopper er d_{ij} og π_{ij} i den sidste matrice sat korrekt for alle $v_i, v_j \in V$ (hvis ingen negativ kreds er i grafen).

Bevis: Invarianten er, at $D^{(k)}$ indeholder længden af korteste vej mellem v_i og v_j som kun passerer knuderne $v_1, v_2, \ldots, v_k$ (udover endepunkterne v_i og v_j). Vises ved induktion på k.

Johnsons algoritme [1977]

Bruger:

- Kører Bellman-Ford-Moore én gang på let udvidet graf.
- Herudfra justering af kantvægte så alle bliver positive uden essentielt at ændre korteste veje (se lemma på næste side).
- Kører Dijkstra fra alle knuder.

Kører i $O(nm \log n + nm) = O(nm \log n)$ tid, klarer negative vægte.

Re-weighting

Se på situationen, hvor vi til alle knuder $v \in V$ tildeler et tal $\phi(v)$.

Ud fra ϕ kan vi lave nye vægte $\tilde{w}$ i grafen på følgende måde:

$$\tilde{w}(u, v) = w(u, v) + \phi(u) - \phi(v).$$

Re-weighting

Se på situationen, hvor vi til alle knuder $v \in V$ tildeler et tal $\phi(v)$.

Ud fra ϕ kan vi lave nye vægte $\tilde{w}$ i grafen på følgende måde:

$$\tilde{w}(u, v) = w(u, v) + \phi(u) - \phi(v).$$

Se på en sti $v_1, v_2, \ldots, v_k$. Da gælder

$$\begin{aligned} \sum_{i=1}^{k-1} \tilde{w}(v_i, v_{i+1}) &= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + \phi(v_i) - \phi(v_{i+1})) \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + (\phi(v_1) - \phi(v_k)) \end{aligned}$$

da summen er teleskoperende.

Re-weighting

Se på situationen, hvor vi til alle knuder $v \in V$ tildeler et tal $\phi(v)$.

Ud fra ϕ kan vi lave nye vægte $\tilde{w}$ i grafen på følgende måde:

$$\tilde{w}(u,v) = w(u,v) + \phi(u) - \phi(v).$$

Se på en sti $v_1, v_2, \ldots, v_k$. Da gælder

$$\begin{aligned} \sum_{i=1}^{k-1} \tilde{w}(v_i, v_{i+1}) &= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + \phi(v_i) - \phi(v_{i+1})) \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + (\phi(v_1) - \phi(v_k)) \end{aligned}$$

da summen er teleskoperende.

Med andre ord er stilængden under de nye vægte lig stilængden under de gamle, med en additiv korrektion baseret på stiens endepunkter.

Dvs. denne korrektion er *den samme* for alle stier fra s $(= v_1)$ til t $(= v_k)$.

Re-weighting

Se på situationen, hvor vi til alle knuder $v \in V$ tildeler et tal $\phi(v)$.

Ud fra ϕ kan vi lave nye vægte $\tilde{w}$ i grafen på følgende måde:

$$\tilde{w}(u,v) = w(u,v) + \phi(u) - \phi(v).$$

Se på en sti $v_1, v_2, \ldots, v_k$. Da gælder

$$\begin{aligned}
\textstyle\sum_{i=1}^{k-1} \tilde{w}(v_i, v_{i+1}) &= \textstyle\sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + \phi(v_i) - \phi(v_{i+1})) \\
&= \textstyle\sum_{i=1}^{k-1} w(v_i, v_{i+1}) + (\phi(v_1) - \phi(v_k))
\end{aligned}$$

da summen er teleskoperende.

Med andre ord er stilængden under de nye vægte lig stilængden under de gamle, med en additiv korrektion baseret på stiens endepunkter.

Dvs. denne korrektion er *den samme* for alle stier fra s $(= v_1)$ til t $(= v_k)$.

Så (d)en korteste sti fra s til t er *den samme sti* under både w og $\tilde{w}$.

Endvidere er der en negativ kreds under w hvis og kun hvis der en negativ kreds under $\tilde{w}$ (da $v_k = v_1$ i en kreds, så $\phi(v_k) - \phi(v_1) = 0$).

A^* [Hart, Nilsson, Raphael, 1968]

A^*-algoritmen kan ses som en tunings-metode til Dijkstra for det (ofte forekommende) tilfælde, at man søger efter sti fra s til en *specifik* målknude t:

Ny ingrediens: Forsøg til alle knuder v at lave en *gæt* $h(v)$ på den korteste afstand fra v til t, dvs. et gæt på $\delta(v, t)$. Man kalder også $h(v)$ for en *heuristik*.

A^* [Hart, Nilsson, Raphael, 1968]

A^*-algoritmen kan ses som en tunings-metode til Dijkstra for det (ofte forekommende) tilfælde, at man søger efter sti fra s til en *specifik* målknude t:

Ny ingrediens: Forsøg til alle knuder v at lave en *gæt* $h(v)$ på den korteste afstand fra v til t, dvs. et gæt på $\delta(v,t)$. Man kalder også $h(v)$ for en *heuristik*.

Intuition: hvis $v.d$ (som i Dijkstra, når v udtages af PQ) er lig $\delta(s,v)$, da er $v.d + h(v)$ et gæt på $\delta(s,v) + \delta(v,t)$, hvilket er længden af den korteste vej fra s til t gennem v.

A^* [Hart, Nilsson, Raphael, 1968]

A^*-algoritmen kan ses som en tunings-metode til Dijkstra for det (ofte forekommende) tilfælde, at man søger efter sti fra s til en *specifik* målknude t:

Ny ingrediens: Forsøg til alle knuder v at lave en *gæt* $h(v)$ på den korteste afstand fra v til t, dvs. et gæt på $\delta(v, t)$. Man kalder også $h(v)$ for en *heuristik*.

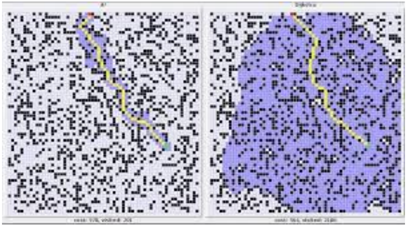
Intuition: hvis $v.d$ (som i Dijkstra, når v udtages af PQ) er lig $\delta(s, v)$, da er $v.d + h(v)$ et gæt på $\delta(s, v) + \delta(v, t)$, hvilket er længden af den korteste vej fra s til t gennem v.

Idé: gå frem som i Dijkstra (inkl. samme update af $v.d$-værdier), men lad nøgle i PQ være $v.d + h(v)$.

Dvs. udvid søgning via knuder, som *gættes* at være på *korteste sti fra s til t*. Til sammenligning kan Dijkstra siges at udvide via knuder, som *vides* at være de *nærmeste til s*.

A^* i praksis

Eksempel med grid-baseret graf. Knuder = de hvide grid-celler, kanter med længde én mellem hvide naboceller. Heuristikken $h(v)$ er lig Euklidisk afstand (fugleflugt) fra celle v til målcellen t.



Dijkstra (højre figur): Undersøger jævnt i alle retninger.

A^* med ovenstående heuristik (venstre figur): Undersøger mere mod målet. Færre knuder besøges, derfor hurtigere i praksis.

Korrekthed og worst case køretid af A^*?

En heuristik kaldes konsistent hvis der for alle knuder v og alle kanter (v, u) i v's naboliste gælder:

$$h(v) \leq w(v, u) + h(u)$$

Dvs. at heuristikkens gæt (på korteste vej til t) for v's naboer ikke er i modstrid med heuristikkens gæt (på korteste vej til t) for v.

Korrekthed og worst case køretid af A^*?

En heuristik kaldes konsistent hvis der for alle knuder v og alle kanter (v, u) i v's naboliste gælder:

$$h(v) \leq w(v, u) + h(u)$$

Dvs. at heuristikkens gæt (på korteste vej til t) for v's naboer ikke er i modstrid med heuristikkens gæt (på korteste vej til t) for v.

Man kan vise, at med en konsistent heuristik er A^* det samme som Dijkstra på en graf med justerede vægte.

Heraf kan man vise korrekthed (at den korteste vej mellem s og t returneres af A^*), og at worst case køretiden er lig worst case køretiden for Dijkstra.