Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

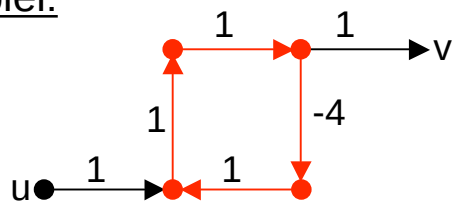
Graphen mit negativen Gewichten (1)

Existenz kürzester Wege für beliebige Graphen ist nicht gesichert:

Hat ein Graph auch negative Gewichte, dann ist die Existenz eines kürzesten Weges nicht gesichert, da Zyklen mit negativer Länge existieren können.

Durch mehrfaches Ablaufen solcher Zyklen kann die Weglänge beliebig klein (negativ) werden.

Beispiel:



Zyklus mit Länge = 1 + 1 – 4 + 1 = -1

Damit kann die Weglänge von u nach v beliebig klein werden.

Existenz kürzester, einfacher Wege ist dagegen gesichert:

- Die Eindeutigkeit von einfachen Wegen (d.h. Knoten dürfen nicht mehrfach auftreten) ist gegeben.
  Begründung: es gibt zwischen zwei Knoten nur endlich viele einfache Wege
- Für die Bestimmung kürzester, einfacher Wege mit beliebigen Gewichten sind bis heute keine effizienten Algorithmen bekannt.

Graphen mit negativen Gewichten (2)

Einschränkung auf gewichtete Digraphen ohne Zyklen negativer Länge

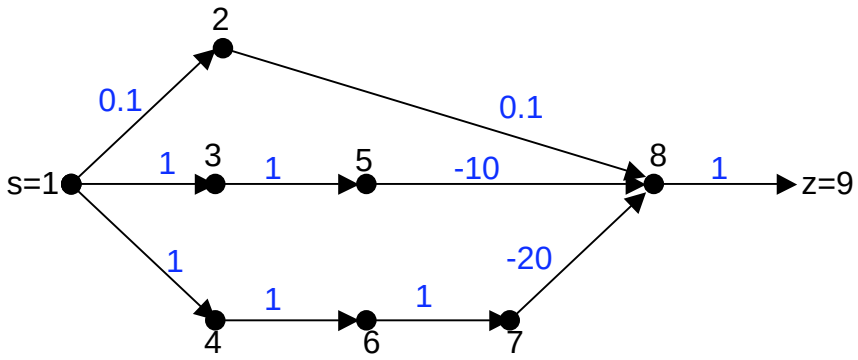
- Da es nur für Graphen ohne Zyklen negativer Länge effiziente Algorithmen gibt, wollen wir uns auf solche Graphen einschränken.
- Beachte, dass sich in ungerichteten Graphen mit negativen Kanten trivialerweise Zyklen negativer Länge ergeben (negative Kante mehrfach hin- und zurücklaufen).

  Da andererseits ungerichtete Graphen mit nur positiven Gewichten mit dem Algorithmus von Dijkstra schon befriedigend behandelt werden können, wollen wir uns im folgenden nur mit Digraphen (gerichtete Graphen) beschäftigen.

Aufgabe 2.2

a) Finden Sie den kürzesten Weg von s nach z.

b) Zeigen Sie, dass der Algorithmus von Dijkstra nicht den kürzesten Weg findet.



Algorithmus von Moore und Ford (1)

Problem:

Eingabe: Gewichteter Digraph (Gewichte können negativ sein) mit Startknoten s.
Ausgabe: Kürzeste Wege zwischen s und allen anderen Knoten, falls Graph keine Zyklen negativer Länge hat.
Ansonsten erfolgt eine entsprechende Ausgabe.

Idee:

- Ausgangspunkt ist der Algorithmus von Dijkstra.
- Beim Algorithmus von Dijkstra wird jeder Knoten v genau einmal besucht und zwar dann, wenn der kürzeste Weg zu v bekannt ist.

  Kommen aber negative gewichtete Kanten vor, dann muss jeder Knoten im allgemeinen mehrere Male besucht werden und zwar immer dann, wenn sich ein noch kürzerer Weg ergibt.
- Die Reihenfolge in der die Knoten besucht werden, spielt nun keine Rolle mehr.

  Es wird daher eine Schlange statt einer Vorrangswarteschlange gewählt.

Algorithmus von Moore und Ford (2)

```
void shortestPath  (Vertex s, Graph G,  int d[ ], Vertex p[ ])
{
        Queue kl<Vertex>;

        for (jeden Knoten v) {
            d[v] = ∞;
            p[v] = undef;
        }
        d[s] = 0;  // Startknoten
        kl.push(s);

        while (! kl.empty())
        {
            v = kl.top(); kl.pop();
            for (jeden Nachfolger w von v ) {
(1)             if (d[v] + c(v,w) < d[w]) {
(2)                 p[w] = v;
(3)                 d[w] = d[v] + c(v,w);
(4)                 if (! kl.isElement(w) )
(5)                     kl.push(w);
                }
            }
        }
}
```

Eingabe: Startknoten s und gewichteter Digraph G

Ausgabe: Distanzfeld d und Vorgängerfeld p

In der Kandidatenliste sind alle Knoten abgespeichert, die als nächstes besucht werden müssen.

Die Zugriffsoperationen auf die Kandidaten-liste sind blau gekennzeichnet.

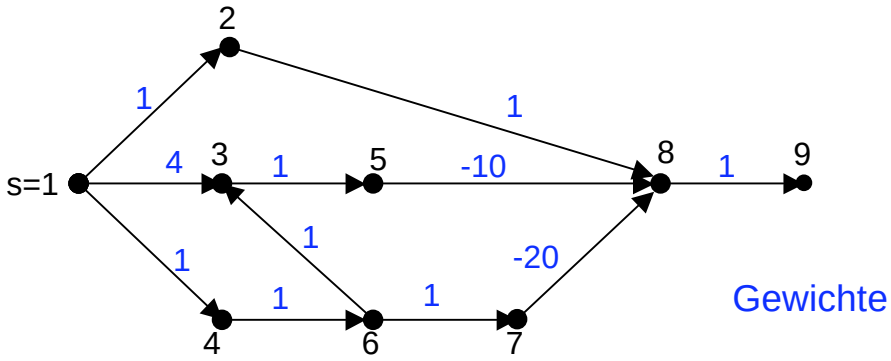
Distanzwert für w verbessert sich. Kürzester Weg nach w geht nun über v.

w muss nochmals besucht werden.

Algorithmus von Moore und Ford (3)

Aufgabe 2.3

Welche Werte nimmt die Kandidatenliste und das Distanz- und Vorgängerfeld im Algorithmus von Moore und Ford ein, wenn er auf folgenden Graphen mit Startknoten s angesetzt wird.



Algorithmus von Moore und Ford (4)

Analyse

- Enthält der Graph keine Zyklen mit negativer Länge, dann wird jeder Knoten maximal (|V|-1)-mal in die Schlange kl eingefügt (Zeile (5) im Algorithmus).
  (Begründung siehe [Turau]).

  Wird ein Knoten |V|-mal eingefügt, dann muss es ein Zyklus mit negativer Länge geben und es könnte im Algorithmus eine entsprechende Ausgabe erfolgen.
- Aufwand:

  Würde jeder Knoten genau einmal in die Schlange kl eingefügt werden, wäre der Aufwand für alle for-Schleifen gerade O(|E|). Denn jede Kante im Graph wird genau einmal in einer der for-Schleifen betrachtetet.

  Da nun aber jeder Knoten (|V|-1)-mal in die Schlange kl eingefügt wird,
  erhält man insgesamt:

  T = O(|E|*|V|).

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Netzpläne (1)

Definition von Netzplänen:

Netzpläne sind gewichtete, azyklische Digraphen mit genau einem Start und einem Endknoten. Die Knoten (außer Start- und Endknoten) stellen oft gewisse Aktivitäten dar und sind mit Gewichten (z.B. Zeitangaben) versehen.

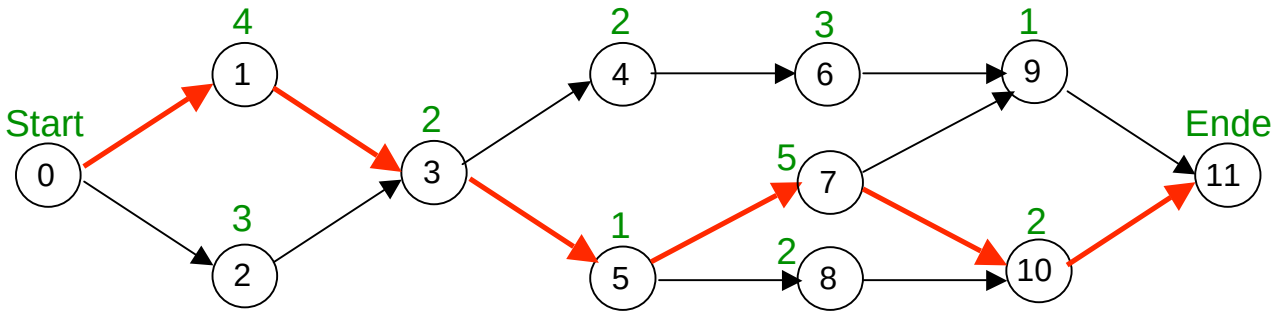
Kritischer Pfad in Netzplänen

Kritischer Pfad ist ein Weg vom Start- zum Ziel-Knoten mit maximalen Zeitaufwand.

Beispiel: Projektplan

Die Knoten sind Tätigkeiten, die in einem Projekt durchgeführt werden müssen.
Die Gewichte geben die Zeitdauer der Aktivitäten an.
Die gerichteten Kanten legen Reihenfolgenbeziehungen fest.

Die Länge eines kritischen Pfades gibt die frühestmögliche Projektfertigstellung an.



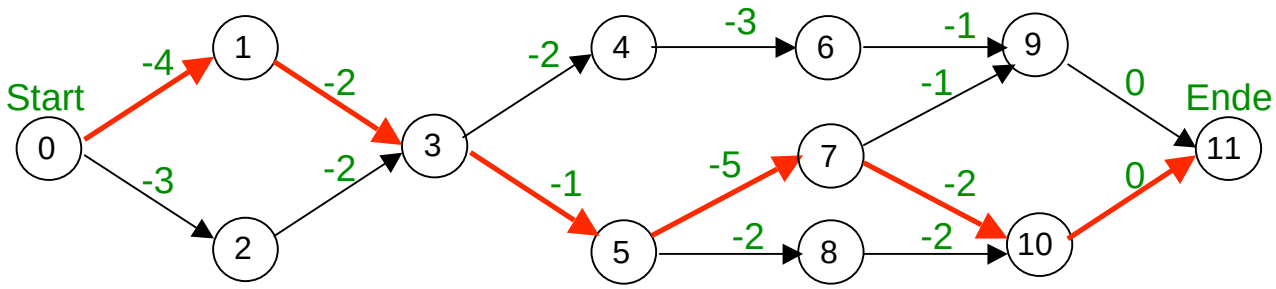
Netzplan und kritischer Pfad mit Zeitaufwand 14

Netzpläne (2)

Bestimmung eines kritischen Pfads

Werden die an den Knoten notierten Gewichte an die einmündenden Kanten mit umgekehrtem Vorzeichen „verschoben“, erhält man einen gewichteten, azyklischen Graphen.

Die Bestimmung des kritischen Pfades lässt sich dann zurückführen auf die Berechnung eines kürzesten Weges vom Start- zum Zielknoten.



Gewichteter, azyklischer Digraph mit kürzestem Weg der Länge -14.

Algorithmus

Prinzipiell ließe sich der kritische Pfad mit dem Algorithmus von Moore und Ford bestimmen.

Für den Spezialfall der Netzpläne ist jedoch ein auf toplogischer Sortierung basierender Algorithmus wesentlich effizienter.

Erweiterte topologische Sortierung (1)

Problem:

Eingabe: Gewichteter, azyklischer Digraph (Gewichte können negativ sein) mit genau einem Startknoten s.

Ausgabe: Kürzeste Wege zwischen s und allen anderen Knoten, falls Graph keine Zyklen negativer Länge hat. Ansonsten erfolgt eine entsprechende Ausgabe.

Idee: Topologische Sortierung mit Abspeichern der gefundenen Wege und Distanzen

- Besuche die Knoten wie bei der topologischen Sortierung. Dazu werden in der Schlange nur solche Knoten gehalten, deren Vorgänger bereits besucht worden sind.
- Bei jedem Knotenbesuch werden dann wie beim Algorithmus von Moore und Ford für alle Nachfolgerknoten die Distanzen und der gefundene Weg gegebenenfalls aktualisiert. (siehe Zeilen (1), (2) und (3) im Algorithmus von Moore und Ford)
- Aufwand:

  Wie bei der topologischen Sortierung ergibt sich:

  T = O(|E|+|V|).

Erweiterte topologische Sortierung (2)

```
void void allShortestPath (Graph G, int d[ ], Vertex p[ ])
{
    int inDegree[n];   // n ist Anz. der Knoten in G
    Queue<Vertex> q;

    for (jeden Knoten v) {
        d[v] = ∞;
        p[v] = undef;
    }

    for (jeden Knoten v) {
        inDegree[v] = Anzahl der Vorgänger;
        if (inDegree[v]  == 0) {
            q.push(v);
            d[v] = 0;
        }
    }

    if (q.size() > 1)
        cout << “Fehler; mehr als ein Startknoten”;
```

Ausgabe:
Distanzfeld d und Vorgängerfeld p

Eingabe:
Gewichteter, azyklischer Digraph G mit genau einem Startknoten.

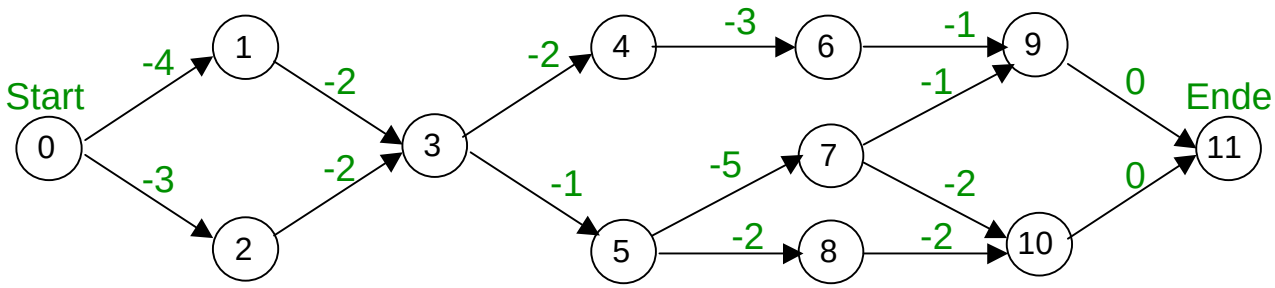
```
    int k = 0;

    while (! q.empty() ) {
        v = q.front(); q.pop(); k++;
        for (jeden Nachfolger w von v ) {
            if (d[v] + c(v,w) < d[w]) {
                p[w] = v;
                d[w] = d[v] + c(v,w);
            }
            if(--inDegree[w] == 0)
                q.push(w);
        }
    }

    if (k != n)
        cout << “Fehler;”
            << “Graph ist zyklisch ”;
}
```

Programmteile, um die die toplogische Sortierung erweitert wurde, sind blau gekennzeichnet.

Erweiterte topologische Sortierung (3)

Aufgabe 2.4

Welche Werte nimmt die Schlange und das Distanz- und Vorgängerfeld im Algorithmus mit erweiterter topologischer Sortierung ein, wenn er auf folgenden Graphen angesetzt wird.



Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
  - Problemstellung
  - Ungewichtete Graphen
  - Distanzgraphen
  - Gewichtete Digraphen
  - Netzpläne
  - Alle kürzeste Wege in gewichteten Digraphen
- Minimal aufspannende Bäume
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Algorithmus von Floyd (1)

Problem:

Eingabe: Gewichteter Digraph (Gewichte können negativ sein)

Ausgabe: Kürzeste Wege zwischen allen Knotenpaaren,
falls Graph keine Zyklen negativer Länge hat.
Ansonsten erfolgt eine entsprechende Ausgabe.

Optimalitätsprinzip:

Der Algorithmus baut im wesentlichen auf das Optimalitätsprinzip von Bellman auf, das für Graphen ohne Zyklen negativer Länge gilt.

Sei u ein beliebiger Knoten auf einem kürzesten Weg von v nach w.
Dann gilt:

$$d(v,w) = d(v,u) + d(u,w).$$

D.h. Teilwege von kürzesten Wegen müssen ebenfalls kürzeste Wege sein.

Algorithmus von Floyd (2)

Idee:

- Die Knoten seien nummeriert: $V = \{v_1, v_2, \ldots, v_n\}$.
- Definiere:

  $D^k(i, j) =$ Distanz von v_i nach v_j, wobei nur Wege betrachtet werden, deren Knoten (außer Start- v_i und Endknoten v_j) aus $\{v_1, v_2, \ldots, v_k\}$ sind.
- Für k = 0 erhält man gerade die Gewichtsmatrix:

  $D^0(i, j) = c(v_i, v_j)$ (1)
- Für k > 0 lässt sich aus dem Optimalitätsprinzip folgende Rekursionsformel herleiten:

  $D^k(i, j) = \min \{ D^{k-1}(i, j) , D^{k-1}(i, k) + D^{k-1}(k, j) \}$ (2)

Länge des kürzesten Wegs von v_i nach v_j, ohne über v_k zu gehen.

Länge des kürzesten Wegs von v_i nach v_j über v_k setzt sich zusammen aus kürzestem Weg von v_i nach v_k und kürzestem Weg v_k nach v_j.

Algorithmus von Floyd (3)

Idee (Fortsetzung):

- $D^k(i, j)$ lässt sich für jedes k durch eine 2-dimensionale Matrix $D^k[i][j]$ darstellen.
  D^0 wird mit Hilfe von (1) berechnet; D^1 ergibt sich aus D^0, D^2 aus D^1, usw.
  mit Hilfe von (2). D^n ist die Lösung unseres Problems.
- D^k wird aus D^{k-1} berechnet. Man käme für die Berechnung von D^k mit 2 Matrizen aus:
  eine Matrix für den alten Wert D^{k-1}und eine Matrix für neuen Wert D^k.
- Man kommt jedoch auch mit genau einer Matrix aus!
  Es gilt nämlich:

$$D^k(i, j) = \min \{ D^{k-1}(i, j) , D^{k-1}(i, k) + D^{k-1}(k, j) \}$$
$$= \min \{ D^{k-1}(i, j) , D^k(i, k) + D^k(k, j) \} \quad (3)$$

  Denn $D^k(i, k) = D^{k-1}(i, k)$ und $D^k(k, j) = D^{k-1}(k, j)$. (d.h. Wege mit v_k als Anfangs- oder Endpunkt bleiben in einer Iteration unverändert).

  Damit kann (3) durch folgende Programmanweisung berechnet werden:

```
D[i][j]  = min {D[i][j] , D[i][k] + D[k][j] }
```

Neuer Wert von D[i][j] ist $D^k(i, j)$	Alter Wert von D[i][j] ist $D^{k-1}(i, j)$

Algorithmus von Floyd (4)

Idee (Fortsetzung):

- Damit läßt sich D[i][j] wie folgt berechnen:

```
// Starte mit D^0:
for (int i = 1; i <= n; i++) {
      D[i][i] = 0;
      for (int j = 1; j <= n; j++)
            D[i][j] = c(v_i, v_j);
}
for (int k = 1; k <= n; k++)
      // Berechne D^k:
      for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                  if (D[i][j] > D[i][k] + D[k][j])
                        D[i][j] = D[i][k] + D[k][j]);
```

D[i][j] = min {D[i][j] , D[i][k] + D[k][j] };

- Zur Speicherung der gefundenen Wege wird eine Vorgängermatrix P eingesetzt: P[i][j] = Vorgängerknoten von v_j im kürzesten Weg von v_i nach v_j.

- Damit kann der kürzeste Weg von s nach z ausgegeben werden (in umgekehrter Reihenfolge):

```
v = z; cout << v <<endl;
while ( v != s ) {
      v = P[s][v]; cout << v << endl;
}
```

Algorithmus von Floyd (5)

```
void allShortestPath (Graph G,  int D[ ][ ], Vertex P[ ][ ])
{
    // Initialisiere D und P:
    for (int i = 1; i <= n; i++) {
        D[i][i] = 0;
        for (int j = 1; j <= n; j++) {
            D[i][j] = c(v_i, v_j);
            if (D[i][j] != ∞) // Kante (v_i,v_j) definiert
                P[i][j] = i;
            else
                P[i][j] = 0; // kein Vorgänger
        }
    }
    for (int k = 1; k <= n; k++)
    // Berechne D^k:
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                if (D[i][j] > D[i][k] + D[k][j]) {
                    D[i][j] = D[i][k] + D[k][j]);
                    P[i][j] = P[k][j];
                }
}
```

Ausgabe:
Distanz- D und
Vorgängermatrix P

Eingabe:
gewichteter Digraph G
mit n Knoten.

Algorithmus von Floyd (6)

Analyse

- Enthält der Graph Zyklen mit negativer Länge, dann wird ein Diagonalelement von D negativ.

  Der Algorithmus kann leicht um eine passende Abprüfung mit entsprechender Ausgabe erweitert werden.

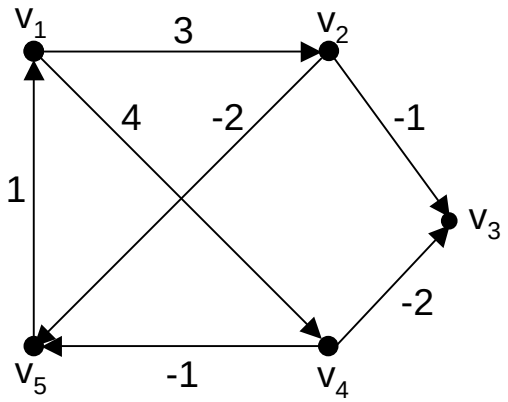
- Aufwand:

  Die dreifach geschachtelten for-Schleifen ergeben:

  $$\mathbf{T = O(|V|^3).}$$

Algorithmus von Floyd (7)

Beispiel (aus [Turau]):



D^0

0	3	∞	4	∞
∞	0	-1	∞	-2
∞	∞	0	∞	∞
∞	∞	-2	0	-1
1	∞	∞	∞	0

D^1

0	3	∞	4	∞
∞	0	-1	∞	-2
∞	∞	0	∞	∞
∞	∞	-2	0	-1
1	4	∞	5	0

D^2

0	3	2	4	1
∞	0	-1	∞	-2
∞	∞	0	∞	∞
∞	∞	-2	0	-1
1	4	3	5	0

D^3

0	3	2	4	1
∞	0	-1	∞	-2
∞	∞	0	∞	∞
∞	∞	-2	0	-1
1	4	3	5	0

D^4

0	3	2	4	1
∞	0	-1	∞	-2
∞	∞	0	∞	∞
∞	∞	-2	0	-1
1	4	3	5	0

D^5

0	3	2	4	1
-1	0	-1	3	-2
∞	∞	0	∞	∞
0	3	-2	0	-1
1	4	3	5	0

P^0

0	1	0	1	0
0	0	2	0	2
0	0	0	0	0
0	0	4	0	4
5	0	0	0	0

P^1

0	1	0	1	0
0	0	2	0	2
0	0	0	0	0
0	0	4	0	4
5	1	0	1	0

P^2

0	1	2	1	2
0	0	2	0	2
0	0	0	0	0
0	0	4	0	4
5	1	2	1	0

P^3

0	1	2	1	2
0	0	2	0	2
0	0	0	0	0
0	0	4	0	4
5	1	2	1	0

P^4

0	1	2	1	2
0	0	2	0	2
0	0	0	0	0
0	0	4	0	4
5	1	2	1	0

P^5

0	1	2	1	2
5	0	2	1	2
0	0	0	0	0
5	1	4	0	4
5	1	2	1	0