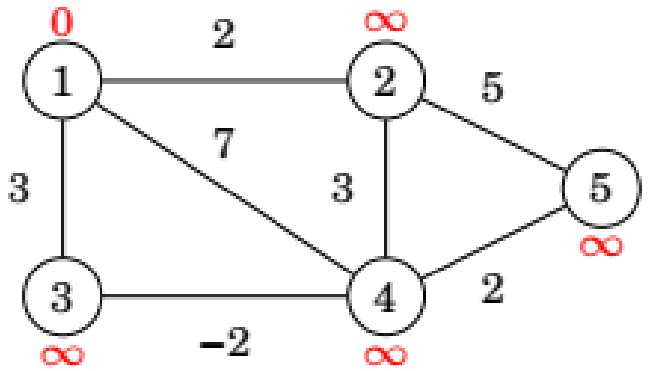
Shortest Path

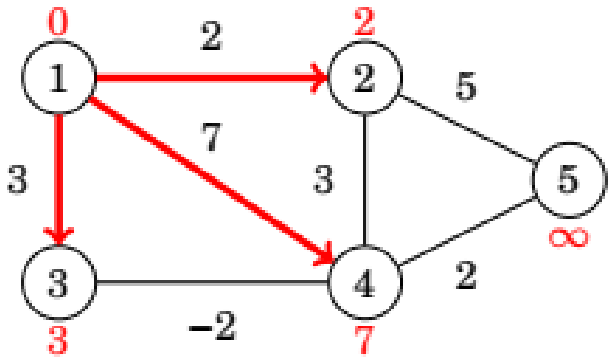
- การหาเส้นทางที่สั้นที่สุดระหว่างโหนดสองโหนดในกราฟเป็นปัญหาสำคัญที่มี application ในชีวิตประจำวันใช้มากมาย ตัวอย่างเช่นปัญหาที่เกี่ยวกับ เครือข่ายถนนจะคำนวณความยาวของ shortest possible length ระหว่างเมืองสองเมืองเมื่อกำหนดความยาวของถนนมาให้
- ในกราฟแบบไม่มีน้ำหนัก ความยาวของเส้นทางเท่ากับจำนวนของเส้นเชื่อม ดังนั้นเราจึงสามารถใช้ breadth–first search ในการหา shortest path
- อย่างไรก็ตาม ในเรื่องนี้เราจะสนใจกราฟแบบมีน้ำหนักเสนออัลกอริทึมในการหา shortest paths.

Bellman–Ford algorithm

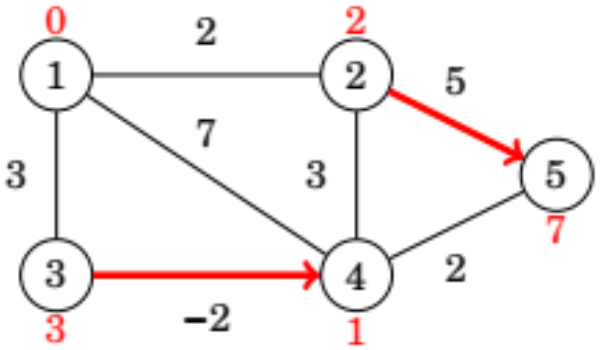
- **Bellman–Ford algorithm** หา shortest paths จากโหนดเริ่มต้นไปยังทุกโหนดในกราฟ อัลกอริทึมสามารถประมวลผลกราฟได้ทุกรูปแบบ ทั้งนี้กราฟต้องไม่มี cycle ที่มีความยาวเป็นลบ (cycle with negative length)
- ถ้ากราฟมี negative cycle, อัลกอริทึมสามารถตรวจสอบได้
- อัลกอริทึมจะเก็บระยะทางจากโหนดเริ่มต้นไปยังทุกโหนดในกราฟ เริ่มต้น ระยะทางจากโหนดเริ่มต้นเป็น 0 และระยะทางไปยังทุกโหนดเป็น infinite
- อัลกอริทึมจะลดระยะทางโดยหาเส้นเชื่อมที่ทำให้เส้นทางสั้นลง จนกระทั่งไม่สามารถลดระยะทางได้อีกแล้ว



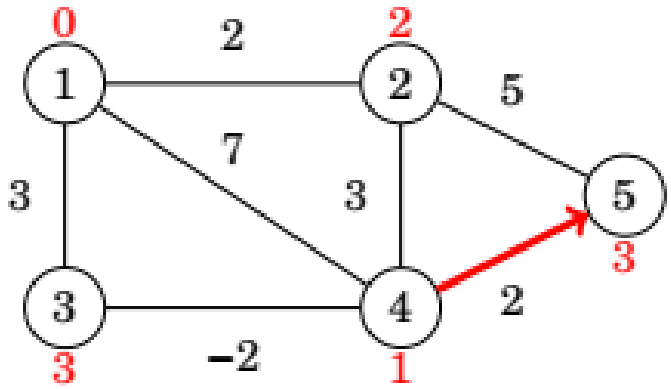
- แต่ละโหนดในกราฟจะถูกกำหนดระยะทาง เริ่มต้นระยะทางของ start node เป็น 0 และระยะทางไปยังโหนดอื่นๆ ทุกโหนดเป็น infinite
- อัลกอริทึมจะค้นหาเส้นเชื่อมที่ลดระยะทางได้ ขั้นแรกทุกเส้นเชื่อมจากโหนด 1 จะลดระยะทาง



- หลังจากนั้น เส้นเชื่อม 2 –> 5 และ 3 –> 4 จะลดระยะทาง

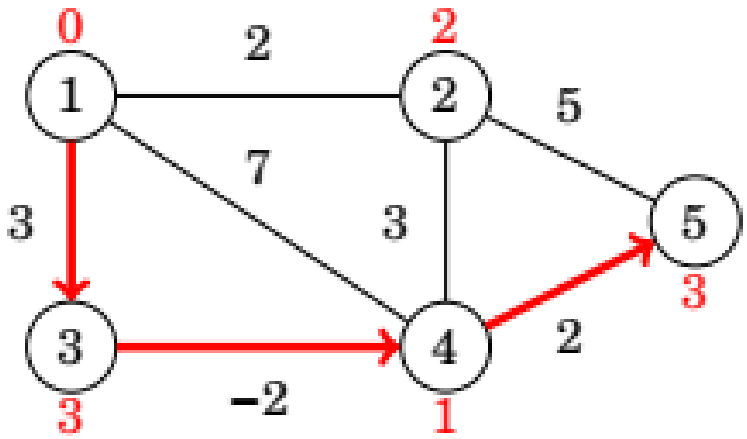


- สุดท้ายมีการปรับอีกครั้ง



- หลังจากนี้ ไม่มีเส้นเชื่อมที่สามารถลดระยะทางได้
- นั่นหมายความว่านี่เป็นระยะทางสุดท้าย และเราได้คำนวณระยะทางสั้นที่สุดจากโหนดเริ่มต้นไปยังทุกโหนดแล้ว

- ตัวอย่าง ระยะทางที่สั้นที่สุดจากโหนด 1 ไปโหนด 5 มีค่าเป็น 3 ซึ่งสอดคล้องกับเส้นทางที่ได้



Implementation

- การ implement ต่อไปของ Bellman–Ford algorithm จะตัดสินระยะทางที่สั้นที่สุดจากโหนด x ไปยังทุกโหนดในกราฟ
- สมมติว่ากราฟถูกเก็บด้วยวิธี edge list
- edges ประกอบด้วย tuples ในรูป (a, b, w), หมายความว่ามีเส้นเชื่อมจาก a ไป b ด้วยน้ำหนัก w
- อัลกอริทึมใช้เวลาทำงาน n–1 รอบ ในแต่ละรอบนั้นจะพิจารณาทุกเส้นเชื่อมและพยายามลดระยะทางจาก x ไปยังทุกโหนดในกราฟ
- ค่าคงที่ INF แทนระยะทางเป็น infinite

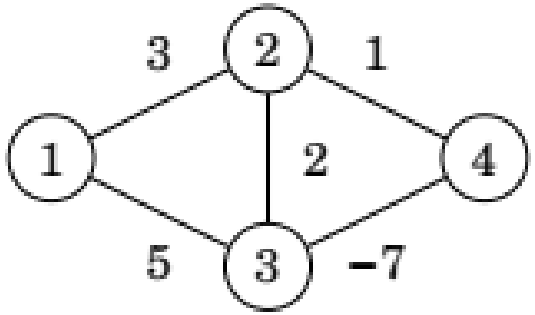
```
for (int i = 1; i <= n; i++)
  distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
  for (auto e : edges) {
    int a, b, w;
    tie(a, b, w) = e;
    distance[b] = min(distance[b], distance[a]+w);
  }
}
```

- เวลาในการทำงานของ algorithm เป็น O(nm) เพราะว่าอัลกอริทึมทำงาน n–1 รอบและแต่ละรอบตรวจสอบทุกเส้นเชื่อม m เส้น
- ถ้าไม่มี negative cycles ในกราฟ ทุกระยะทางจะเสร็จสิ้นหลังจาก n–1 รอบเพราะว่า shortest path แต่ละเส้นมีเส้นเชื่อมได้ไม่เกิน n–1 เส้น

- ในทางปฏิบัติแล้วระยะทางสุดท้ายมักจะเสร็จก่อน n–1 รอบ
- ดังนั้นสามารถเขียน code ดักว่าถ้าไม่มีอะไรเปลี่ยนแล้วหยุดการทำงานก่อนได้เพื่อลดจำนวนรอบในการทำงาน

Negative cycle

- Bellman–Ford algorithm สามารถใช้ตรวจสอบว่ากราฟมี cycle ที่มีความยาวเป็นลบได้ ตัวอย่างเช่นกราฟ



- มี negative cycle 2 –> 3 –> 4 –> 2 ที่มีความยาวเป็น –4
- ถ้ากราฟมี negative cycle เราสามารถทำให้สั้นลงได้เรื่อยๆ
- ดังนั้น shortest path จะไม่มีความหมายในเหตุการณ์นี้

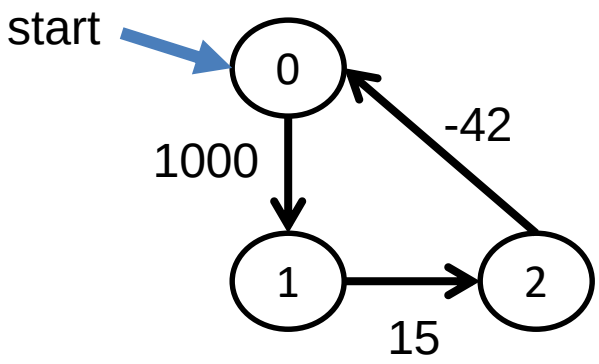
- Negative cycle สามารถถูกตรวจสอบได้จาก Bellman–Ford algorithm โดยการรันอัลกอริทึม *n* รอบ
- ถ้ารอบสุดท้ายระยะทางลดลงได้อีกแสดงว่ากราฟมี negative cycle
- สังเกตว่าอัลกอริทึมนี้สามารถถูกใช้เพื่อหา negative cycle โดยไม่สนใจว่าโหนดเริ่มต้นเป็นโหนดใดได้

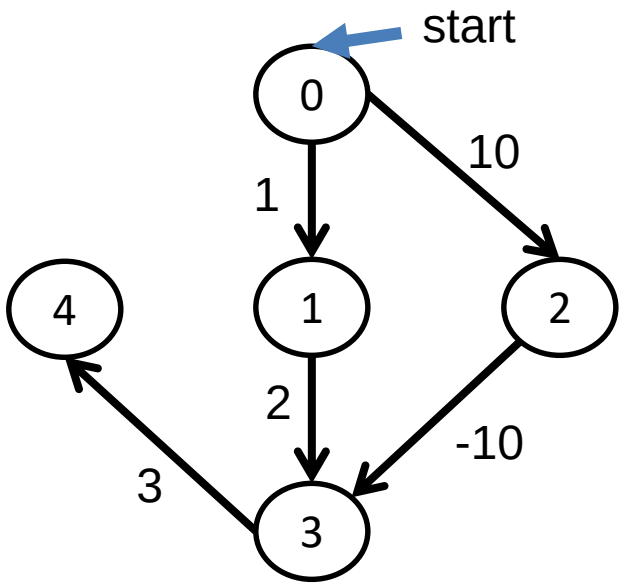
```cpp
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;
  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));
  }
```

```
// Bellman Ford routine
 vi dist(V, INF); dist[s] = 0;
 for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times, overall O(VE)
   for (int u = 0; u < V; u++)                          // these two loops = O(E)
     for (int j = 0; j < (int)AdjList[u].size(); j++) {
       ii v = AdjList[u][j];        // we can record SP spanning here if needed
       dist[v.first] = min(dist[v.first], dist[u] + v.second);         // relax
     }

 bool hasNegativeCycle = false;
 for (int u = 0; u < V; u++)                          // one more pass to check
   for (int j = 0; j < (int)AdjList[u].size(); j++) {
     ii v = AdjList[u][j];
     if (dist[v.first] > dist[u] + v.second)                  // should be false
       hasNegativeCycle = true;     // but if true, then negative cycle exists!
   }
 printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

 if (!hasNegativeCycle)
   for (int i = 0; i < V; i++)
     printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

 return 0;
}
```

แบบ Adjlist https://bit.ly/2WmlAjF

- ทดลองกราฟนี้



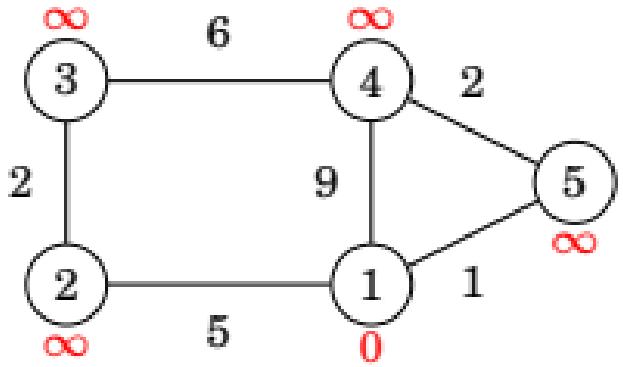


Dijkstra's algorithm

- Dijkstra's algorithm จะหาเส้นทางที่สั้นที่สุดจากโหนดเริ่มต้นไปยังทุกโหนดในกราฟเช่นเดียวกับ Bellman–Ford algorithm
- ข้อดีของ Dijkstra's algorithm คือมันมีประสิทธิภาพมากกว่าและสามารถใช้กับกราฟขนาดใหญ่ได้ อย่างไรก็ตาม algorithm นี้จะทำงานได้ต้องไม่มี negative weight ในกราฟ
- เช่นเดียวกับ Bellman–Ford algorithm Dijkstra's algorithm นั้นเก็บระยะทางที่ไปยังโหนดและลดค่ามันระหว่างการค้นหา
- Dijkstra's algorithm มีประสิทธิภาพเพราะว่ามันประมวลผลเส้นเชื่อมแต่ละเส้นในกราฟเพียงครั้งเดียว โดยใช้ความจริงที่ว่ากราฟไมมี negative edges

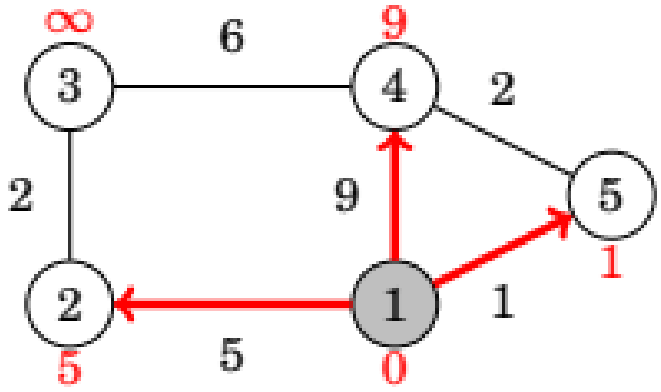
ตัวอย่าง

- พิจารณากราฟต่อไปนี้เราจะใช้ Dijkstra's algorithm เมื่อเริ่มต้นที่โหนด 1



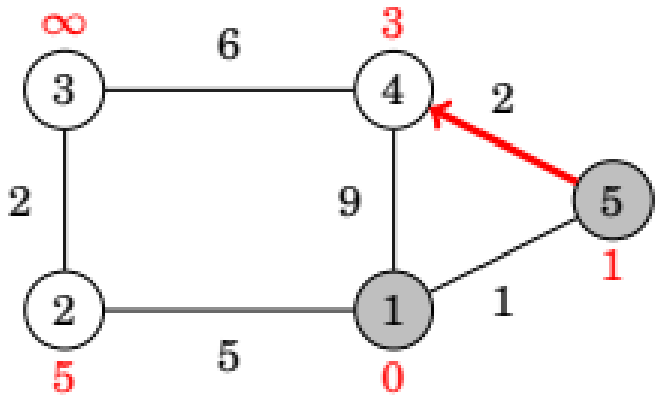
- เช่นเดียวกับ Bellman–Ford algorithm เริ่มต้นจะกำหนดระยะทางจากโหนดเริ่มต้นเป็น 0 และ โหนดที่เหลือเป็น infinite
- ในแต่ละรอบ Dijkstra's algorithm จะเลือกโหนดที่ยังไม่ถูกพิจารณาและมีระยะทางใกล้ที่สุด
- ในตัวอย่างโหนดแรกเป็นโหนด 1 ซึ่งมีระยะทางเป็น 0

- เมื่อโหนดถูกเลือก algorithm จะพิจารณาทุกเส้นเชื่อมที่เริ่มต้นที่โหนดนั้นและลดระยะทาง

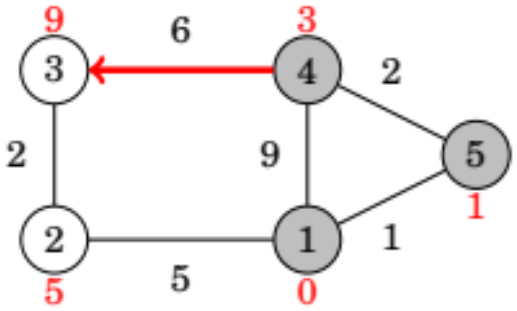


- ในกรณีนี้เส้นเชื่อมจากโหนด 1 ลดระยะทางของโหนด 2, 4 และ 5 ซึ่งระยะทางปัจจุบันเป็น 5, 9 และ 1

- โหนดต่อไปที่จะประมวลผลคือโหนด 5 ที่มีระยะทางเป็น 1 ซึ่งจะไปลดระยะทางที่ไปยังโหนด 4 จาก 9 เป็น 3

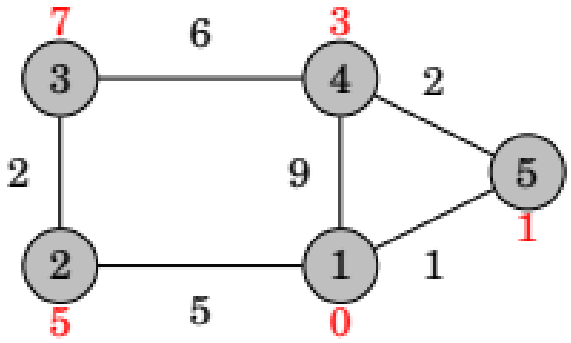


- หลังจากนั้นโหนดต่อไปคือโหนด 4 จะลดระยะทางจาก 3 เป็น 9



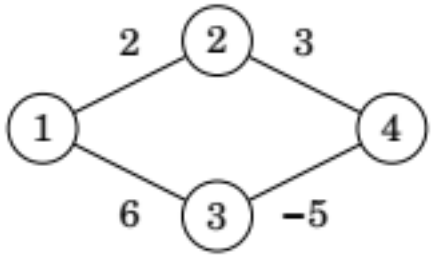
- คุณสมบัติของ Dijkstra's algorithm คือเมื่อโหนดถูกเลือกแล้ว ระยะทางของมันจะถือว่าสิ้นสุดแล้ว
- ตัวอย่างเช่นที่จุดนี้ ระยะทาง 0 1 และ 3 เป็นระยะทางสุดท้ายของโหนด 1, 5 และ 4

- หลังจากนั้น algorithm จะทำงานกับ 2 โหนดที่เหลือ ซึ่งได้ระยะทางสุดท้ายดังรูป



Negative edges

- ประสิทธิภาพของ Dijkstra's algorithm นั้นอยู่บนพื้นฐานของความจริงที่ว่าในกราฟไม่มีเส้นเชื่อมที่มีน้ำหนักเป็นลบ (negative edges)
- ถ้ามีเส้นเชื่อมเป็นลบ algorithm อาจจะให้ผลลัพธ์ที่ผิดได้ ตัวอย่างเช่น กราฟต่อไปนี้



- ระยะทางสั้นที่สุดจาก 1 ไปโหนด 4 คือ 1->3->4 ซึ่งความยาวเป็น 1 อย่างไรก็ตาม Dijkstra's algorithm หาเส้นทาง 1->2->4 ตามที่ค่าน้ำหนักน้อยที่สุดของเส้นเชื่อม

Implementation

- ต่อไปเป็น implementation ของ Dijkstra's algorithm ซึ่งคำนวณระยะทางสั้นที่สุดจากโหนด x ไปยังโหนดที่เหลือในกราฟ
- Dijkstra's algorithm จะเก็บ priority queue สมมติว่าชื่อ pq ที่เก็บ pair ข้อมูลของโหนด
  - ข้อมูลตัวแรกเป็นระยะทางจากโหนดไปยัง source
  - ข้อมูลตัวที่สองเป็นหมายเลขโหนด
- การเก็บแบบนี้ เราจะมา sort pq ตามระยะห่างจาก source จากน้อยไปมากได้ซึ่งถ้าเท่ากัน เราก็จะใช้หมายเลขโหนดแทน
- ทั้งนี้สามารถ implement ด้วย data structure ตัวอื่นได้ด้วยเช่นใช้ binary heap แต่ว่าไม่มีใน builtin library

- pq เริ่มต้นจะเก็บ 1 item นั่นคือ base case (0,s) ซึ่งเป็นจริงเพราะว่าเป็น source vertex ระยะทางก็เป็น 0 นั่นเอง

- จากนั้น Dijkstra's algorithm จะทำซ้ำตามนี้จน pq หมด
  - มันจะหยิบข้อมูลจาก pq pair(d,u) จากตัวหน้าสุดของ pq เป็นการเลือกแบบ greedy
  - ถ้าระยะทางของ u จาก source ที่ถูกบันทึกไว้ใน d มากกว่าระยะทาง dist[u] ก็จะไม่ทำอะไรกับ u แต่ถ้าไม่ใช่ เราก็จะปรับปรุงค่าของ u

- เมื่ออัลกอนี้ประมวลผลโหนด u มันจะพยายามปรับให้ลดลงของทุกเพื่อนบ้าน v ของ u (relax เป็นการเซตค่า dist[v] = min(dist[v],dist[u]+w(u,v)))

- ทุกครั้งที่มัน relax เส้นเชื่อม u–>v มันจะ enqueue pair(ระยะทางอันใหม่ที่สั้นกว่าของ v จาก source, v) เข้าไปใน pq และทิ้ง pair(ระยะทางอันเก่าที่ยาวกว่าของ v จาก source, v) ออกจาก pq

- วิธีการนี้เรียกว่า lazy deletion สิ่งนี้เป็นสาเหตุให้มีมากกว่า 1 copy ของโหนดเดียวกันใน pq ทีมีระยะต่างกันจาก source ดังนั้นเราต้องเชคก่อนเพื่อที่จะประมวลผล pair ของข้อมูลโหนดที่ถูก dequeue ในครั้งแรก ซึ่งจะทำให้ได้ระยะทางที่ถูกและสั้น (ระยะทางของ copy อื่นนั้นจะไม่ทันสมัยและยาวกว่า

- ซึ่ง code ต่อไปนี้ก็จะคล้าย BFS อยู่

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;
  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii());
// assign blank vectors of pair<int, int>s to AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));
// directed graph
 }
```

```
 vi dist(V, INF);
  dist[s] = 0;                          // INF = 1B to avoid overflow
  priority_queue< ii, vector<ii>, greater<ii> > pq;
  pq.push(ii(0, s));
                               // ^to sort the pairs by increasing distance from s
  while (!pq.empty()) {                                              // main loop
    ii front = pq.top();
    pq.pop();      // greedy: pick shortest unvisited vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;            // this check is important, see the explanation
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];                          // all outgoing edges from u
        if (dist[u] + v.second < dist[v.first]) {
          dist[v.first] = dist[u] + v.second;                // relax operation
          pq.push(ii(dist[v.first], v.first));
   }
  }
 }
  for (int i = 0; i < V; i++)
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
}
```

https://bit.ly/2HE4RVI

- ทดลองกราฟนี้