Algorithms and Data Structures
Module 3. Dynamic programming

Lecture 15

Dynamic programming for calculating distances in graphs. Part 2.

Dijkstra’s algorithm

Let $\delta(i, v)$ denote the minimum weight of a path from s to v which contains at most i edges.

$$\delta(i,v) = \begin{cases} 0, & if\ v = s\ and\ i = 0 \\ \infty, & if\ v \neq s\ and\ i = 0 \\ \min\left[\begin{matrix} \delta(i-1,v), \\ \min\limits_{(u,v)\in E}\{\delta(i-1,u) + w(u,v)\} \end{matrix}\right], & otherwise \end{cases}$$

Dijkstra’s algorithm

The pseudocode of the algorithm:

```
// Vertices are identified with
// their indices, 0..$n$-1
Create matrix d[0..$n$-1].
// Initialization
d[s] = 0
for v = 0 to n-1:
    if v != s then d[v] = ∞
```

Dijkstra’s algorithm

```
// Filling the table
for i=1 to n-1:
    for each edge (u,v):
        if d[u]+w[u,v]<d[v]
            then d[v]=d[u]+w[u,v]
```

Time complexity: $O(nm), n = |V|, m = |E|$.

Further issues

In the next lecture we will explore more issues related to shortest path problem:

- Dijkstra’s algorithm version for the case of non-negative edge weights (based on the ‘best-first’ graph traversal).
- Building the shortest paths, in addition to the distances.
- Problem 3 (all-to-all shortest paths problem).

Dijkstra’s algorithm: non-negative edges

Let us see at the Dijkstra’s algorithm for the general case.

```
// Filling the table
for i=1 to n-1:
    for each edge (u,v):
        if d[u]+w[u,v]<d[v]
            then d[v]=d[u]+w[u,v]
```

For the case of non-negative edges, we can organize calculations in a way that each edge is processed at most once.

Dijkstra’s algorithm: non-negative edges

For the case of non-negative edges, we can organize calculations in a way that each edge is processed at most once.

In order to get such improvement we need to analyze and process vertices in the order of increasing their distances from s. We select an unprocessed vertex v with the minimum tentative distance from s and build the shortest path to v by augmenting the path to some other previously processed vertex (the *predecessor* of v).

Dijkstra’s algorithm: non-negative edges

The initialization is essentially the same:

```
// Vertices are identified with
// their indices, 0..n-1
Create matrix d[0..n-1].
// Initialization
d[s] = 0
for v = 0 to n-1:
    if v != s then d[v] = ∞
```

Dijkstra’s algorithm: non-negative edges

But we will use a priority queue similar to BFS. The keys will be the tentative distances from s to all other vertices

```
for v = 0 to n-1: Enqueue(v,d[v]);
```

Then we iteratively process vertices; at each iteration we select the vertex with the minimum tentative distance.

Dijkstra’s algorithm: non-negative edges

```
While (Queue is not empty):
    u = GetMin()
    DelMin()
    for each edge (u,v):
        if d[u]+w[u,v]<d[v] then
            d[v]=d[u]+w[u,v]
            ChangePriority(v, d[v])
```

Each vertex is extracted from the priority queue only once. Hence, each edge is processed at most once. Hence, time complexity: $O(m \cdot \log n)$, where m is the quantity of edges, $O(\log n)$ is the complexity of a priority queue operation. Time complexity of the general version: $O(nm)$

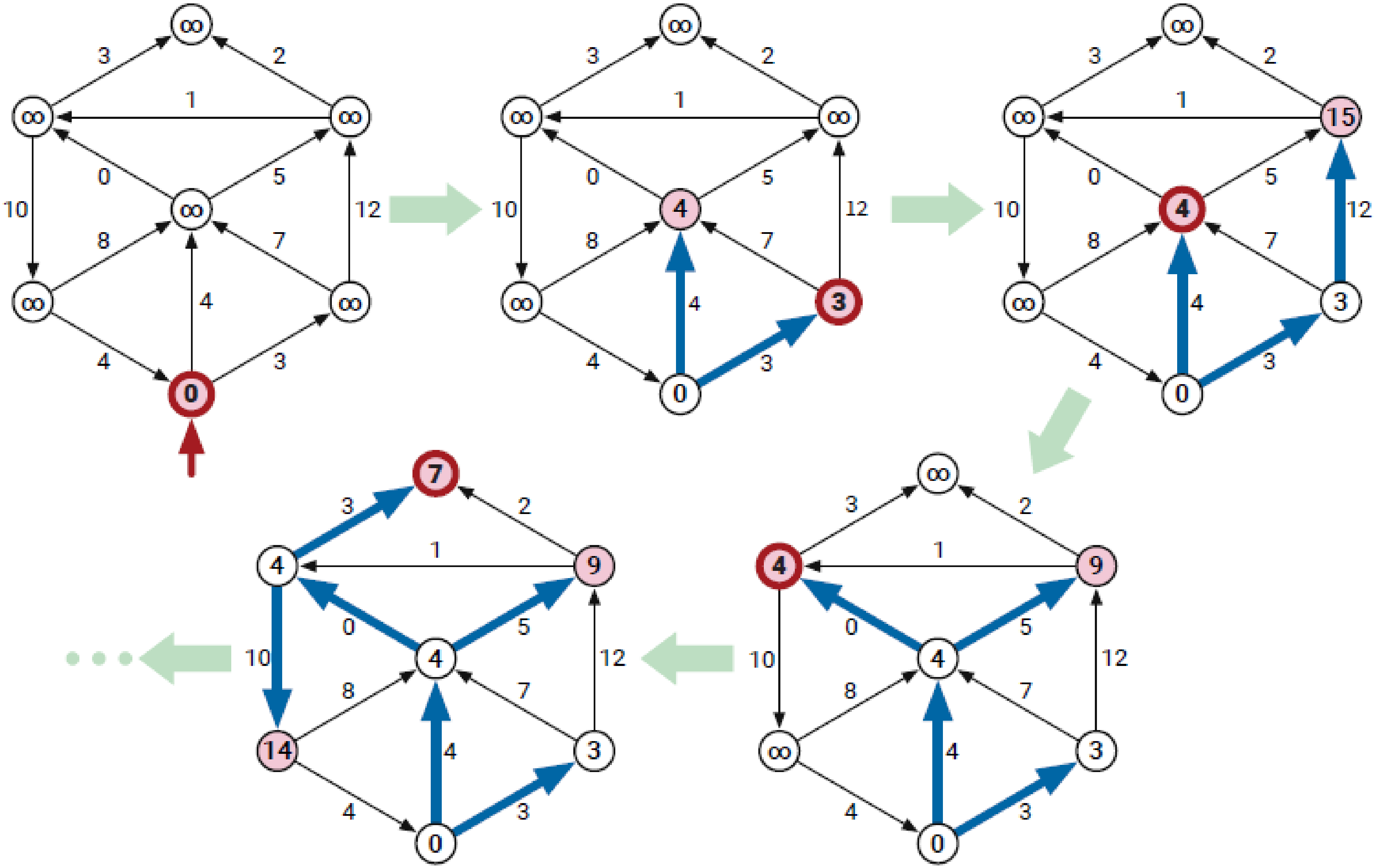
Dijkstra’s algorithm: building paths

Besides calculating distances, for many applications we need to build the shortest paths themselves.

Due to the principle of optimality, the shortest paths from a given source vertex to all other vertices make the *shortest path tree*.

We can build the shortest path to v by augmenting the path to some other previously processed vertex (the *predecessor* of v).

Dijkstra’s algorithm: building paths



Dijkstra’s algorithm: building paths

The principal idea is similar to the BFS-based version of the algorithm: during the run of the algorithm we keep the predecessors for all vertices in an array $p[0..n-1]$.

Vertex u is the predecessor of the vertex v iff we update $d[v]$ while processing the edge (u, v).

Dijkstra’s algorithm: building paths

```
// Vertices are identified with
// their indices, 0..n-1
Create matrices d[0..n-1], p[0..n-1].
// Initialization
d[s] = 0; p[s] = NULL;
for v = 0 to n-1:
    if v != s then
        d[v] = ∞;    p[v] = NULL;
```

Dijkstra’s algorithm: building paths

Dijkstra’s algorithm for the general case:

```
// Filling the table
for i=1 to n-1:
    for each edge (u,v):
        if d[u]+w[u,v]<d[v] then
            d[v]=d[u]+w[u,v];
            p[v] = u;
```

Dijkstra’s algorithm: building paths

Dijkstra’s algorithm for the case of non-negative edges:

```
While (Queue is not empty):
    u = GetMin()
    DelMin()
    for each edge (u,v):
        if d[u]+w[u,v]<d[v] then
            d[v]=d[u]+w[u,v]
            ChangePriority(v, d[v])
            p[v] = u;
```

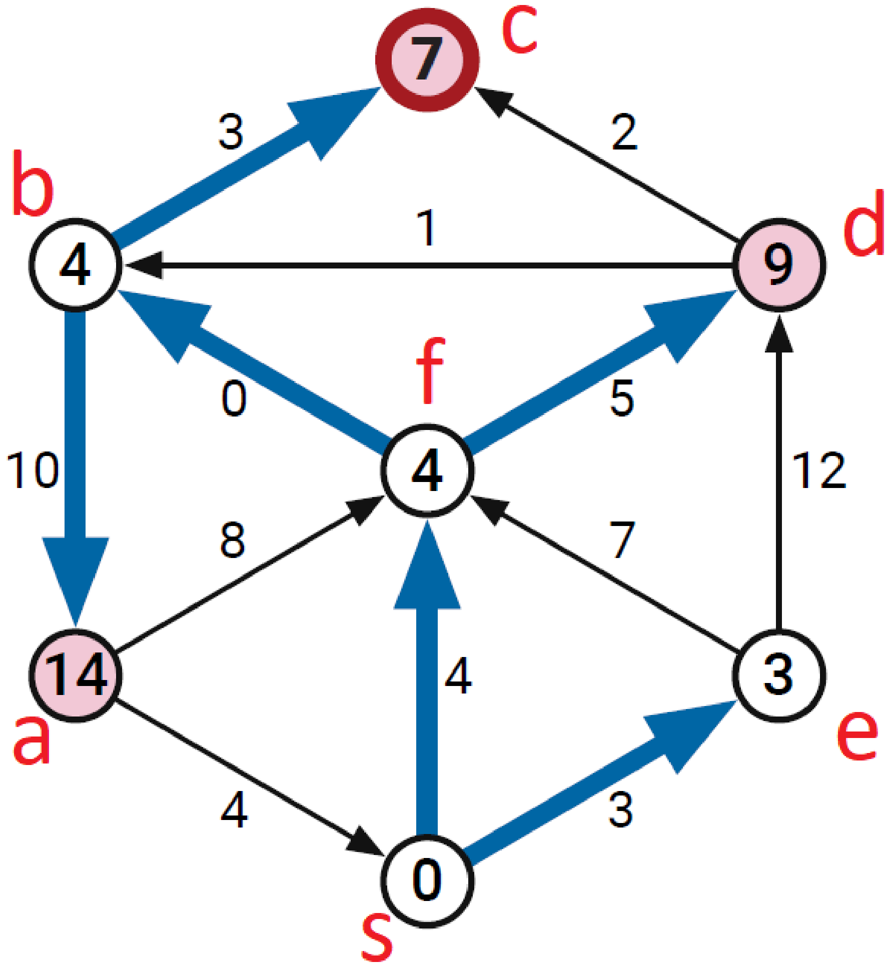
Dijkstra’s algorithm: building paths

Building a shortest path from s to v: start from v and reconstruct the path backward to s. We move from a current vertex u

to $x = p[u]$, then to $y = p[x]$,… , until we get s.

The shortest $s \rightsquigarrow a$ path is:

$$s \longrightarrow f \longrightarrow b \longrightarrow a$$



All-to-all shortest paths problem

Problem 3: Find distances and the shortest paths from s to t for all pairs of vertices.

The result we need: distance matrix $D = \{d[i,j]\}$, where $d[i,j]$ is the distance from i to j.

An obvious way to solve this problem: for each $v \in V$ find the shortest paths from v (as a source vertex s) to all other vertices. The overall complexity:

- $O(nm \cdot \log n)$ for non-negative weights' case;
- $O(n^4)$ for the general case.

All-to-all shortest paths problem

Let us try to apply the dynamic programming approach to this problem. (We still consider the case of graphs without negative cycles.)

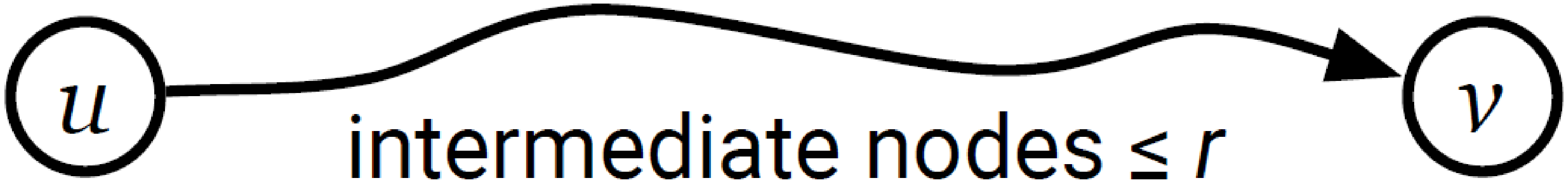
At first we write the recurrence for this problem.

We will apply the approach which differs from that of Dijkstra’s algorithm.

All-to-all shortest paths problem

Let us number the vertices from 1 to n, the order does not matter.

Let $\pi(u, v, r)$ denote the shortest path from u to v that passes through only vertices numbered at most r. That is, the *intermediate* vertices of $\pi(u, v, r)$ should have numbers at most r.

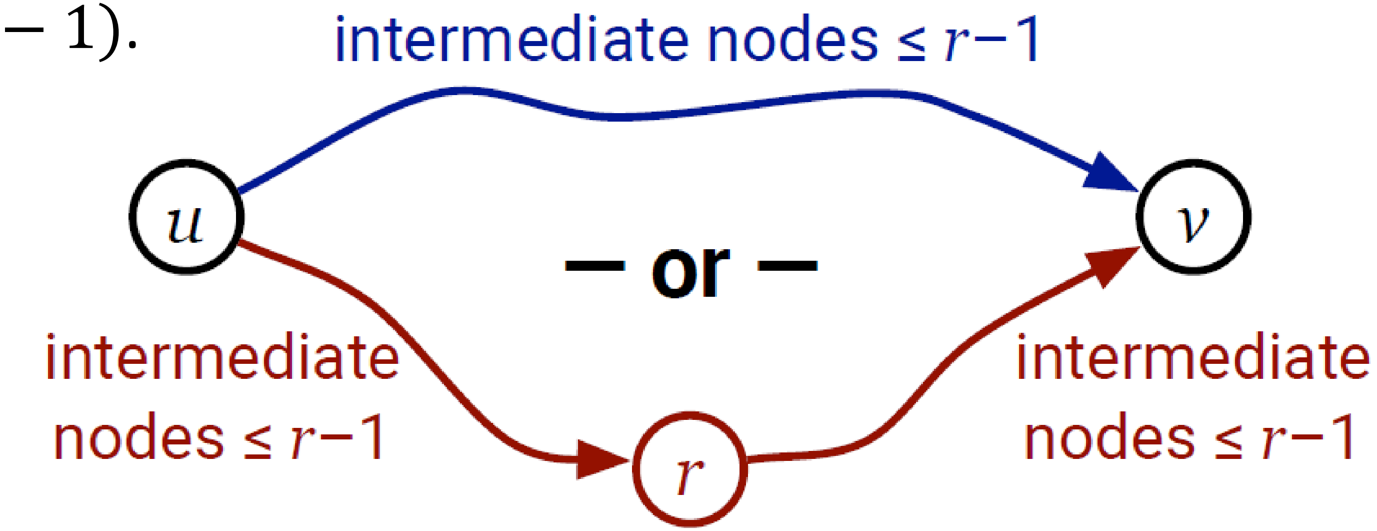


All-to-all shortest paths problem

- The path $\pi(u, v, 0)$ cannot pass through any intermediate vertices, so it must be the edge from *u* to *v*. If u and v are not adjacent, $\pi(u, v, 0)$ is undefined.
- For any integer *r* > 0, either $\pi(u, v, r)$ passes through vertex *r* or it doesn’t.
  - If $\pi(u, v, r)$ passes through vertex *r*, it consists of a subpath from *u* to *r*, followed by a subpath from *r* to *v*. Both of those subpaths pass through only vertices numbered at most $r - 1$. Moreover, those subpaths are as short (have as little weight) as possible with this restriction. So the two subpaths must be $\pi(u, r, r - 1)$ and $\pi(r, v, r - 1)$.

All-to-all shortest paths problem

- For any integer *r* > 0, either $\pi(u, v, r)$ passes through vertex *r* or it doesn't.
  - …
  - On the other hand, if $\pi(u, v, r)$ does not pass through vertex *r*, then it passes through only vertices numbered at most $r - 1$, and it must be the *shortest* path with this restriction. So in this case, we must have $\pi(u, v, r) = \pi(u, v, r - 1)$.



All-to-all shortest paths problem

Hence, the following recurrence holds for the distances:

$$\delta(u,v,r) = \begin{cases} w(u,v), & if\ r = 0 \\ \min\begin{bmatrix} \delta(u,v,r-1), \\ \delta(u,r,r-1) + \delta(r,v,r-1) \end{bmatrix}, & otherwise \end{cases}$$

All we need is to implement this recurrence in code.

All-to-all shortest paths problem

```
// Initialization
for all vertices u:
     for all vertices v:
          d[u,v,0] = w[u,v]
// Fill in matrix D
for r from 1 to $n$:
     for all vertices u:
          for all vertices v:
               if d[u,v,r-1] < d[u,r,r-1]+d[r,v,r-1] then
                    d[u,v,r] = d[u,v,r-1]
               else
                    d[u,v,r] = d[u,r,r-1]+d[r,v,r-1]
```

All-to-all shortest paths problem

```
// Initialization
for all vertices u:
     for all vertices v:
          d[u,v,0] = w[u,v]
// Fill in matrix D
for r from 1 to n:
     for all vertices u:
          for all vertices v:
               if d[u,v,r-1] < d[u,r,r-1]+d[r,v,r-1] then
                    d[u,v,r] = d[u,v,r-1]
               else
                    d[u,v,r] = d[u,r,r-1]+d[r,v,r-1]
```

We do not need the 3rd dimention for D.

The order is arbitrary, in fact.

Floyd-Warshall algorithm

Floyd-Warshall algorithm:

```
// Initialization
for all vertices u:
     for all vertices v:
          d[u,v] = w[u,v]  // We just copy matrix: D = W
// Fill in matrix D
for all vertices r:
     for all vertices u:
          for all vertices v:
               if d[u,v] > d[u,r]+d[r,v] then
                    d[u,v] = d[u,r]+d[r,v]
```

Time complexity: $O(n^3)$.

Floyd-Warshall algorithm: building paths

To build the shortest paths, we trace the maximum number of intermediate vertices on the shortest path: `p[u,v]`.

Update these values every time we update `d[u,v]`.

Floyd-Warshall algorithm

```
// Initialization
for all vertices u:
    for all vertices v:
        d[u,v] = w[u,v]
        p[u,v] = NULL
// Fill in matrices D and P
for all vertices r:
    for all vertices u:
        for all vertices v:
            if d[u,v] > d[u,r]+d[r,v] then
                d[u,v] = d[u,r]+d[r,v]
                p[u,v] = r
```

Floyd-Warshall algorithm

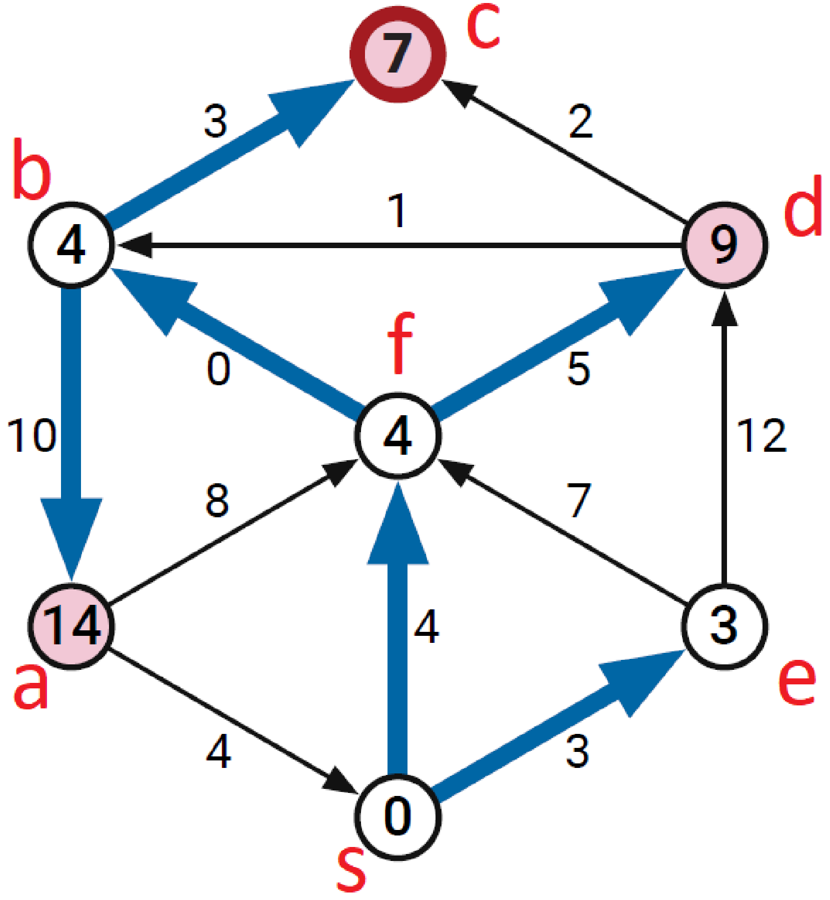
Building a shortest path from u to v: start from the pair of the endpoints: u, v and iteratively fill in the intermediate vertices according to $p[.,.]$.

The process of building the shortest $s \rightsquigarrow a$ path:

$s, a;$ $\quad p[s, a] = f$

$s, f, a;$ $\quad p[s, f] = NULL, p[f, a] = b;$

$s, f, b, a.$


c
7
3
2
b
d
1
4
9
f
0
5
10
4
12
8
7
14
4
3
a
e
4
3
0
s