Shortest Paths in Graphs

Birgit Vogtenhuber

Outline

- Introduction and Definitions
- Algorithm of Dijkstra
- Algorithm of Floyd and Warshall

Motivation and Goal

Many algorithms on graphs are based on the calculation of 'distances' between vertices (examples: driving directions in road networks, number of state transitions between different states of a system).

Distance $d(u, v)$ from $u \in V$ to $v \in V$ in a connected graph $G = (V, E)$:
length of the shortest path from u to v.

- G unweighted: number of edges
- G weighted: sum of edge weights

Graph G can be directed or undirected (or mixed).

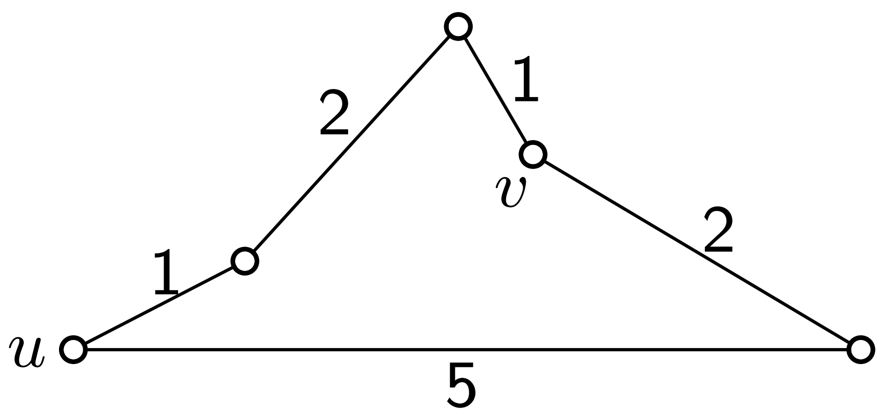
Goal: Compute distances between all pairs of vertices in G.

(Un)weighted Graphs

Question: How can one compute the distances between all pairs of vertices in a connected *unweighted graph* G ?

Using **breadth-first search**, the distance-matrix for a graph G with n vertices and m edges can be computed in $\Theta(n \cdot m)$ time and $\Theta(n^2)$ space.

Question: Does this also work for *weighted graphs* ?



Idea: "Adapt" BFS for shortest paths in weighted graphs.

Dijkstra's Algorithm

Classic shortest path algorithm from Dijkstra [1959]:
For a start vertex s, compute shortest paths from s to all $v \in V$ (tree structure + length).

Question: Why do shortest paths from s to all other vertices form a tree?

Input: A connected graph $G = (V, E, w)$ with non-negative edge weights $w(u, v)$ and a vertex $s \in V$.

Output: The distances $d(s, v)$ in G from s to all vertices $v \in V$ and the tree with the according shortest paths.

Dijkstra's Algorithm

Classic shortest path algorithm from Dijkstra [1959]:
For a start vertex s, compute shortest paths from s to all $v \in V$ (tree structure + length).

Generic step: Given a set T of vertices where for all $v \in T$, $d(s, v)$ is already computed. Choose a vertex $u \in V \setminus T$ whose shortest path from s "found so far" is minimal.

Paths "found so far": paths that only go via vertices in T.

For each vertex v, we maintain:

$L(v)$: length of the shortest path from s to v "found so far".

$pre(v)$: neighbor of v in T via which this shortest path goes.

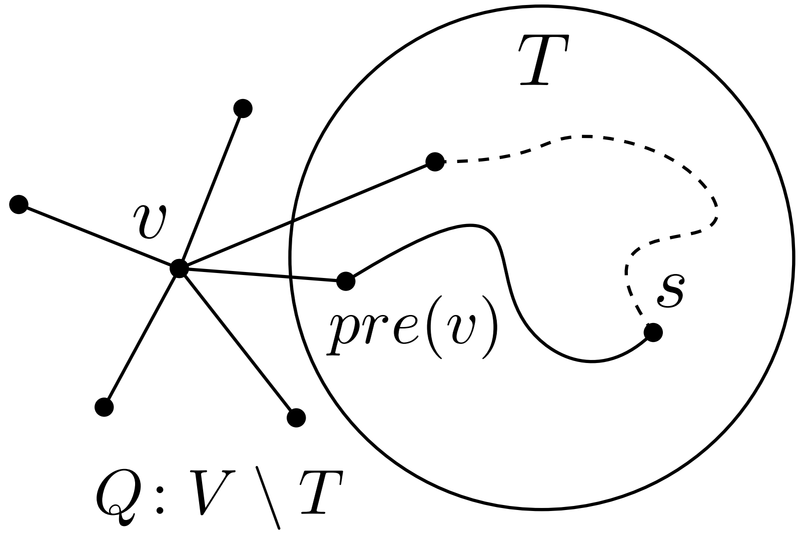
(compare to Prim's MST algorithm)

Dijkstra's Algorithm

Classic shortest path algorithm from Dijkstra [1959]:
For a start vertex s, compute shortest paths from s to all $v \in V$ (tree structure + length).

$$L(v) = \begin{cases} d(s,v) & \text{if } v \in T \\ \infty & \text{if } v \text{ is not adjacent to } T \\ \text{shortest path from } s \text{ to } v \text{ via } T & \text{if } v \notin T,\ v \text{ adjacent to } T \end{cases}$$

A **priority queue** Q contains *all vertices that are not yet in* T, organized by their L-values (for example a min-heap; initially contains all vertices).



Dijkstra's Algorithm

```
for all v ∈ V do L(v) = ∞ od
L(s) = 0; pre(s) = nil
Q = V                                   // build up Q
while Q ≠ 0 do
   u = MIN(Q)
   remove u from Q                      // reorganize Q
   for all v ∈ A(u) do                  // A: adjacency list of G
      if L(v) > L(u) + w(u,v) then
         L(v) = L(u) + w(u,v)           // reorganize Q
         pre(v) = u
      fi
   od
od
```

Dijkstra's Algorithm

Runtime analysis for graph with n vertices and m edges:

- Min-heap with n elements:
  - $\Theta(n)$ time for initialization $Q = V$.
  - $O(\log n)$ time for removal of the minimum.
  - $O(\log n)$ time per update of an L-value.
- Processing vertex u with $\deg(u)$ neighbors:
  removal of u from Q plus $O(\deg(u))$ updated L-values.

$\Rightarrow$ Runtime in total for start vertex s:

$$\Theta(n) + \sum_{u \in V} (1 + \deg(u)) \cdot O(\log n)$$
$$= \Theta(n) + \Theta(n + m) \cdot O(\log n) = O(m \log n),$$

since the graph is connected.

$\Rightarrow$ Computation of distance matrix in $O(nm \log n)$ time.

Dijkstra's Algorithm

Memory analysis for graph with n vertices and m edges:

- $\Theta(n+m) = \Theta(m)$ for G,
- $\Theta(n)$ for Q,
- $\Theta(n)$ for tree T,
- $\Theta(n)$ for lengths L,
- $\Theta(n^2)$ for distance matrix.

$\Rightarrow$ $\Theta(m)$ for shortes path tree and distances from s,
$\Theta(n^2)$ for computing the whole distance matrix.

Dijkstra's Algorithm

Correctness. We will show:

1. For all $v \in T$ we have $L(v) = d(s, v)$.
2. For each $v \notin T$, $L(v)$ is the length of the shortest path from s to v in G that goes only through vertices of T. (or $L(v) = \infty$ if such a path does not exist).

Proof. We use induction on $|T|$.

Induction base: after the first pass, we have
$L(s)=d(s,s)=0$, $L(v)=w(s,v)$ for all $v \in A(s)$,
$L(v)=\infty$ for all $v \notin A(s)$, and $T=\{s\}$.
$\Rightarrow$ Conditions 1. and 2. are fulfilled.

Induction step: u is added to T (and removed from Q).

Dijkstra's Algorithm

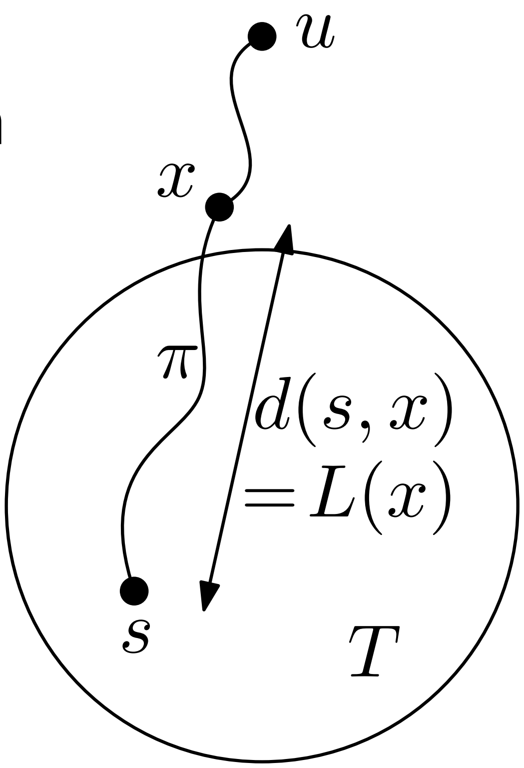
- Assume for a contradiction that $L(u) > d(s,u)$ ($L(u) < d(s,u)$ is impossible) and let π be a shortest path from s to u.

$\Rightarrow$ Since $L(u)$ measures the shortest path from s to u via vertices of T, the path π has vertices outside T.

- Let x be the first vertex on π with $x \notin T$.

$\Rightarrow$ The path from s to x along π is the shortest path from s to x (optimality of partial paths) and goes only via vertices in T.

$\Rightarrow$ $L(x) = d(s,x)$ because of Condition 2.

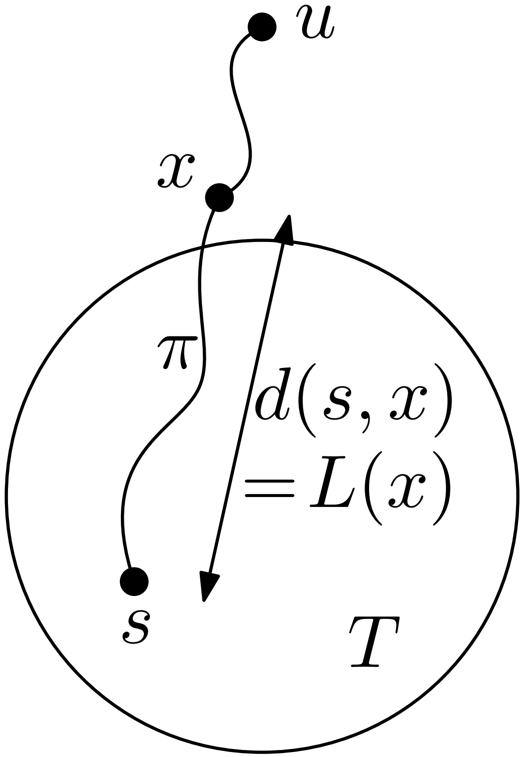


Dijkstra's Algorithm

- $L(x) < L(u)$ because $d(s,x) \leq d(s,u) < L(u)$.
- As both u and x are in Q, this is a contradiction to $L(u) = \min_{v \in Q}\{L(v)\}$.

$\Rightarrow$ $L(u) = d(s,u)$ and hence Condition 1. is maintained when adding u to T.

- Condition 2. is also maintained: When u comes to T, $L(v)$ can only decrease for $v \in A(u)$.



$\Rightarrow$ Dijkstra's algorithm correctly computes the distances from s to all other vertices.

Dijkstra's Algorithm

Remarks:

- Note the similarity of Dijkstra's algorithm with the algorithm of Prim for computing a minimum spanning tree: only the computation of the priorities (p or L) is different.
- For dense graphs ($m = \Theta(n^2)$) the algorithm needs $\Theta(n^3 \log n)$ time to compute the distance matrix.
- If an unsorted list is used for the queue Q, a runtime of $O(\sum_{v \in V} v \in V(n + \deg(v) \cdot 1)) = O(n^2 + m) = O(n^2)$ for start vertex s and $O(n^3)$ for the distance matrix is obtained (independent of m) $\Rightarrow$ good for dense graphs, bad for sparse graphs ($m = \Theta(n)$), works also for Prim.

Dijkstra's Algorithm

Remarks:

Question: We required the input graph G to be connected. Does the algorithm of Dijkstra also work if G is not connected (not every vertex can be reached from every other vertex)?

Question: We required the edge weights $w(u, v)$ in our input graph G to be non-negative. Does the algorithm of Dijkstra also work if edge weights can be negative?

Dijkstra's Algorithm

Remarks:

- The algorithm of Dijkstra does in general not work if some of the edge weights are negative.
- If the graph has a (possibly trivial) cycle with negative length then it's not clear what "shortest path" means (no finite solution minimizes the distance).
- The Bellman-Ford algorithm [1955-1958] can be used for graphs with negative edge weights.
  If a cycle with negative weight can be reached from s, it returns an error. Otherwise the distances from s and a shortest path tree are computed in $O(n \cdot m)$ time.

Floyd-Warshall Algorithm

In the Floyd-Warshall algorithm [1962], the distance matrix is calculated directly. The underlying observations are similar to those in dynamic programming.

Consider a connected weighted graph $G = (V, E, w)$, $V = v_1, ..., v_n$, with non-negative edge weights, and a weight matrix $w(i, j)$, $1 \leq i, j \leq n$, defined by

$$w(i,j) = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

We compute a sequence of matrices $w_1, ..., w_n$ from w with $w_k(i, j) = \min\{w_{k-1}(i, j), w_{k-1}(i, k) + w_{k-1}(k, j)\}$ and $w_0 = w$.

Floyd-Warshall Algorithm

Claim: $w_n(i,j)$ is the distance from v_i to v_j in G.

Proof. We show by induction on k that $w_k(i,j)$ is the length of the shortest path from v_i to v_j via $\{v_1, ..., v_k\}$.

Induction base: For $k = 0$ the statement is true:

- if $i \neq j$ and $v_i v_j \in E$ then $w_0(i,j) = w(v_i, v_j)$;
- if $i \neq j$ and $v_i v_j \notin E$ then $w_0(i,j) = \infty$;
- $w_0(i,i) = 0$.

In all cases, $w_0(i,j)$ is the shortest path from v_i to v_j without intermediate vertices.

Induction step: Assume the statement is correct up to $k-1$ and consider w_k.

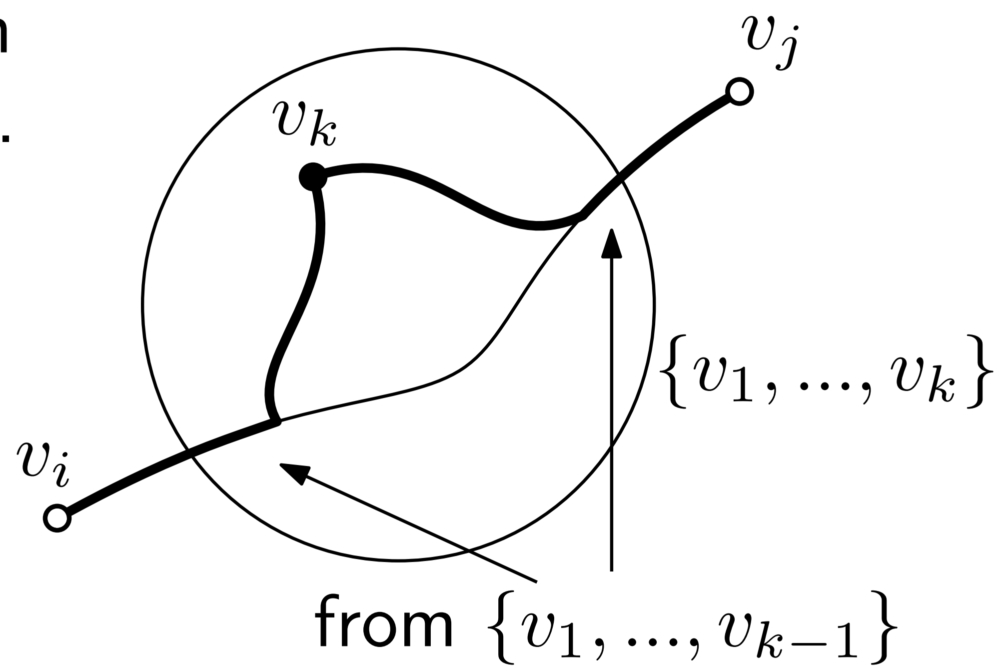
Floyd-Warshall Algorithm

Observation: The shortest path π from v_i to v_j via vertices from $\{v_1, \ldots, v_k\}$ may or may not contain v_k.

- If π contains v_k, then the parts of π from v_i to v_k and from v_k to v_j go only via $\{v_1, \ldots, v_{k-1}\}$.

$\Rightarrow$ By induction, the lengths of those parts are stored in $w_{k-1}(i,k)$ and $w_{k-1}(k,j)$.

$\Rightarrow$ Hence the length of π is $w_{k-1}(i,k) + w_{k-1}(k,j)$.



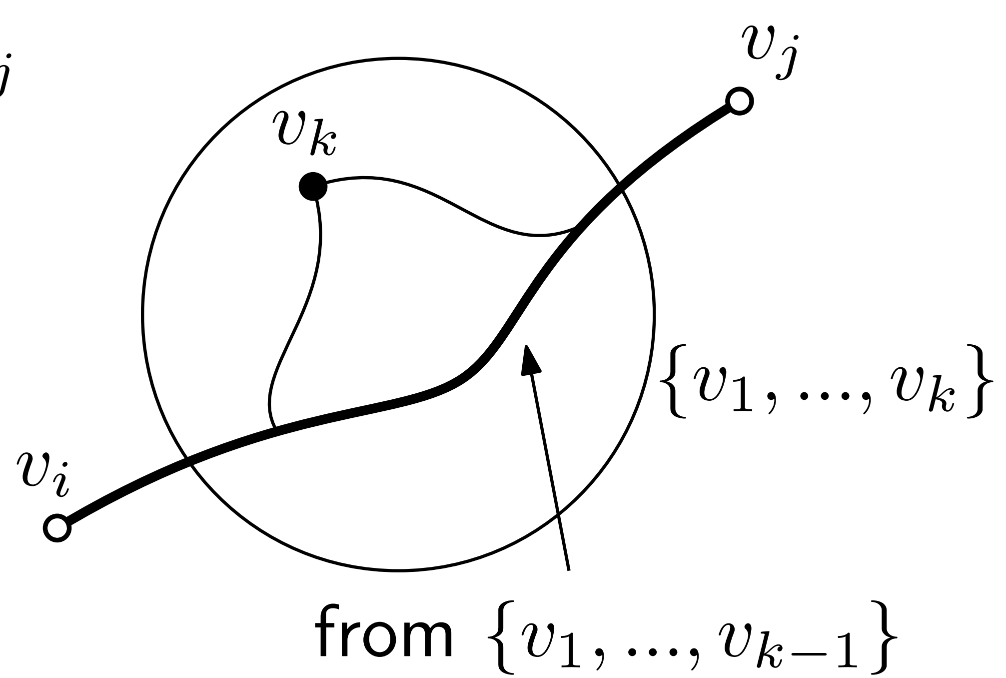
Floyd-Warshall Algorithm

- If π does not contain v_k then π goes via $\{v_1, \ldots, v_{k-1}\}$.

$\Rightarrow$ By induction, the length of π is stored in $w_{k-1}(i,j)$.

- The algorithm takes the minimum of the two considered possibilities $\Rightarrow w_k(i,j)$ is the length of π in both cases.

$\Rightarrow$ $w_n(i,j)$ is the length of the shortest path from v_i to v_j that can go via all vertices of V and hence $w_n(i,j) = d(v_i, v_j)$.



Floyd-Warshall Algorithm

Pseudocode:

$w_0 = w$

for $k = 1$ **to** n **do**
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** n **do**
      $w_k(i,j) = \min\{w_{k-1}(i,j), w_{k-1}(i,k) + w_{k-1}(k,j)\}$
    od
  od
od

Requirements for G with n vertices and m edges:

- Runtime: $\Theta(n^3)$
- Memory: $\Theta(n^2)$

Floyd-Warshall Algorithm

Remarks:

Question: Does the algorithm of Floyd-Warshall work if the input graph is not connected (not every vertex can be reached from every other vertex)?

Question: We required the edge weights $w(u, v)$ in our input graph G to be non-negative. Does the algorithm of Floyd-Warshall work if edge weights can be negative?

Floyd-Warshall Algorithm

Remarks:

- The Floyd-Warshall algorithm also works if the graph is disconnected (if not every vertex can be reached from every other vertex). The distance between such vertices is set to ∞ in the matrix w_n.

- With a small adaption, the Floyd-Warshall algorithm can also be used for graphs with negative edge weights: Then an additional check for the existence of (possibly trivial) cycles with negative length is needed. A graph has a (possibly trivial) cycle with negative length if and only if the matrix w_n contains negative entries in its diagonal.

Conclusion

- Two algorithms for computing all shortest distances between pairs of points in a weighted graph: Dijkstra's algorithm, Algorithm of Flloyd and Warshall
- Animated version of Dijkstra's algorithm available (see animated algorithms webpage)
- Open questions: Discussion session
- Two more questions on shortest paths:
  What about negative edge weights in undirected graphs?
  What about Euclidean shortest paths in complete graphs?

Thank you for your attention.