Fundamental Algorithms
Chapter 9: Weighted Graphs

Jan Křetínský
Winter 2017/18
Weighted Graphs

Definition (Weighted Graph)

A **weighted graph** $G = (V, E)$ is attributed by a function $w$ that assigns a weight $w(e)$ to each edge $e \in E$. 
Weighted Graphs

Definition (Weighted Graph)

A weighted graph $G = (V, E)$ is attributed by a function $w$ that assigns a weight $w(e)$ to each edge $e \in E$.

Comments

- typically: $w(e) > 0$ or $w(e) \geq 0$ (but negative weights possible)
- we will consider weighted graphs with $w : E \rightarrow \mathbb{N}$
- notation: we will also write $w(V, W)$, instead of $w((V, W))$, for the weight $w(e)$ of the edge $e = (V, W)$
- examples: traffic networks, costs for routing, etc.
Shortest Path

Definition (Length of a Path)

The length of a path \( p = (V_0, V_1), (V_1, V_2), \ldots, (V_{n-1}, V_n) \) in a weighted graph is defined as

\[
\overline{w}(p) := \sum_{j=1}^{n} w(V_{j-1}, V_j).
\]
Shortest Path

Definition (Length of a Path)

The length of a path \( p = (V_0, V_1), (V_1, V_2), \ldots, (V_{n-1}, V_n) \) in a weighted graph is defined as

\[
\overline{w}(p) := \sum_{j=1}^{n} w(V_{j-1}, V_j).
\]

Definition (Distance between Vertices)

The distance \( d(V, W) \) between two vertices \( V \) and \( W \) is defined as the length of the shortest path \( p = (V_0, V_1), (V_1, V_2), \ldots, (V_{n-1}, V_n) \) that connects \( V \) and \( W \):

\[
d(V, W) = \min\{ \overline{w}(p) : p = (V_0, V_1), (V_1, V_2), \ldots, (V_{n-1}, V_n), \forall j: (V_{j-1}, V_j) \in E, V = V_0, W = V_n \}\]
All-Pairs Shortest Path

For non-weighted graphs: (try this at home!)
BF-traversal finds the shortest path from a starting node to all connected nodes.
→ is there an efficient algorithm to find the shortest path from all nodes to all other nodes? (“all-pairs shortest path”)

→ is there an efficient algorithm to find which nodes are connected by a path of length $l$?
→ is there an efficient algorithm to find which nodes are connected by only the first $k$ nodes? (assuming an ordering of the nodes)

For weighted graphs:
Generalize the last idea for weighted graphs
→ Incrementally construct shortest paths from nodes connected by only the first $k$ nodes
→ We will implement the algorithm for directed graphs (modifying it for undirected graphs is straightforward)
All-Pairs Shortest Path

For non-weighted graphs: (try this at home!)
BF-traversal finds the shortest path from a starting node to all connected nodes.

→ is there an efficient algorithm to find the shortest path from all nodes to all other nodes? (“all-pairs shortest path”)
→ is there an efficient algorithm to find which nodes are connected by a path of length $l$?
All-Pairs Shortest Path

For non-weighted graphs: (try this at home!)
BF-traversal finds the shortest path from a starting node to all connected nodes.
→ is there an efficient algorithm to find the shortest path from all nodes to all other nodes? (“all-pairs shortest path”)
→ is there an efficient algorithm to find which nodes are connected by a path of length \( l \)?
→ is there an efficient algorithm to find which nodes are connected by only the first \( k \) nodes? (assuming an ordering of the nodes)

For weighted graphs:
Generalize the last idea for weighted graphs
→ Incrementally construct shortest paths from nodes connected by only the first \( k \) nodes
All-Pairs Shortest Path

For non-weighted graphs: (try this at home!)
BF-traversal finds the shortest path from a starting node to all connected nodes.
→ is there an efficient algorithm to find the shortest path from all nodes to all other nodes? (“all-pairs shortest path”)
→ is there an efficient algorithm to find which nodes are connected by a path of length $l$?
→ is there an efficient algorithm to find which nodes are connected by only the first $k$ nodes? (assuming an ordering of the nodes)

For weighted graphs:
Generalize the last idea for weighted graphs
→ Incrementally construct shortest paths from nodes connected by only the first $k$ nodes
→ We will implement the algorithm for directed graphs (modifying it for undirected graphs is straightforward)
Floyd’s Algorithm

Floyd_basic \( (W: \text{Array}[1..n,1..n]) \) {

! Input: weight/adjacency matrix W
! assume: \( W[i,j] = \text{inf} \), if i not connected to j
! Output: \( W[i,j] \) shortest part from i to j

for \( k \) from 1 to n do

! check for all \((i,j)\) whether a shorter path exists
! that runs through vertex \( k \)

for \( i \) from 1 to n do

for \( j \) from 1 to n do

\( W[i,j] = \min(W[i,k]+W[k,j], W[i,j]) \)

end do

end do

end do

}
Floyd’s Algorithm (2)

Disadvantages of Floyd basic:

- input array $W$ is overwritten
- we get the length of the shortest path, but not the path itself!

Floyd ($W$: Array $[1..n,1..n]$,
      $S$: Array $[1..n,1..n]$,
      $P$: Array $[1..n,1..n]$) {
    ! Output: $S$ will contain lengths
    ! $P$ allows to reconstruct shortest path
    for $i$ from 1 to $n$ do
        for $j$ from 1 to $n$ do
            $S[i,j] = W[i,j]$
            $P[i,j] = 0$
        end do
    end do
Floyd’s Algorithm (3)

\[
\text{for } k \text{ from } 1 \text{ to } n \text{ do}
\]
\[
\quad \text{for } i \text{ from } 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \text{for } j \text{ from } 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \quad \text{if } S[i,k] + S[k,j] < S[i,j] \text{ then}
\]
\[
\quad \quad \quad \quad S[i,j] = S[i,k] + S[k,j];
\]
\[
\quad \quad \quad ! \text{ memorize connection via } k
\]
\[
\quad \quad \quad P[i,j] = k;
\]
\[
\quad \quad \text{end if}
\]
\[
\quad \text{end do}
\]
\[
\text{end do}
\]
\[
}\}
\]

Use array P to reconstruct shortest path:

- P[i,j] indicates that shortest path runs through vertex k
- check P[i,k] and P[k,j] for further info
Floyd’s Algorithm – Correctness

Ingredients:

- **Optimality Principle:**
  If the shortest path between nodes $A$ and $B$ visits a node $C$, then this path consists of the shortest paths between $A$ and $C$, and between $C$ and $B$.

- **No cycles:**
  The shortest path between any two nodes does not contain a cycle, i.e., contains any node at most once.
  - while edges are allowed to have negative weights, cycles must not lead to negative weight

- **Loop Invariant** for the k-loop:
  At entry of the k-loop, $S[i,j]$ contains (for every pair $i,j$) the length of the shortest path between $i$ and $j$ that only visits nodes with index smaller than $k$. 
Floyd’s Algorithm on the PRAM

FloydPRAM (W: Array [1 .. n, 1 .. n]) {
    for k from 1 to n do
        for i from 1 to n do in parallel
            for j from 1 to n do in parallel
                if W[i, k] + W[k, j] < W[i, j]
                    then W[i, j] = W[i, k] + W[k, j]
            end do
        end do
    end do
}

Classify concurrent/exclusive read/write?
Floyd’s Algorithm on the PRAM

FloydPRAM (\(W: \text{Array} [1..n, 1..n]\)) {
    for \(k \) from 1 to \(n\) do
        for \(i \) from 1 to \(n\) do in parallel
            for \(j \) from 1 to \(n\) do in parallel
                if \(W[i,k] + W[k,j] < W[i,j]\)
                    then \(W[i,j] = W[i,k] + W[k,j]\)
                end do
            end do
        end do
    end do
}

Classify concurrent/exclusive read/write?

- **concurrent read** to row \(W[*,k]\) and column \(W[k,*]\)
Dijkstra’s Algorithm for Shortest Paths

Problem setting: “single-source shortest path”

- given is a directed graph $G = (V, E)$ and a start vertex $r \in V$
- we want to compute the shortest path from $r$ to each vertex in $G$ that is reachable from $r$
  → this is a spanning tree of shortest paths
Dijkstra’s Algorithm for Shortest Paths

**Problem setting:** “single-source shortest path”
- given is a directed graph $G = (V, E)$ and a start vertex $r \in V$
- we want to compute the shortest path from $r$ to each vertex in $G$ that is reachable from $r$
  → this is a **spanning tree** of shortest paths

**Idea:** “Greedy Algorithm”
- maintain a spanning tree $S$ of vertices and “explored” shortest paths
- maintain a set $Q = V \setminus S$ of unexplored vertices
- for each $v \in Q$, determine the shortest path to $v$ that can be obtained by adding a single edge to the spanning tree $S$
- add $v_{\min}$ (with shortest path) to $S$ and update $Q$
- repeat until all vertices are in the explored path
Dijkstra’s Algorithm – Implementation

**Spanning Tree $S$ of Shortest Paths**

- use an array $\text{Parent}[1..n]$ for the $n$ vertices
- $\text{Parent}[i]$ contains the parent of vertex $i$ in the spanning tree
Dijkstra’s Algorithm – Implementation

Spanning Tree $S$ of Shortest Paths
- use an array $\text{Parent}[1..n]$ for the $n$ vertices
- $\text{Parent}[i]$ contains the parent of vertex $i$ in the spanning tree

Set $Q$ of Unexplored Vertices
- accompanied by an array $\text{Dist}[1..n]$
- $\text{Dist}[i]$ contains the shortest path to vertex $i$ that adds only one edge to $S$
- we will need to update $\text{Dist}[1..n]$ after each change of $Q$
- for vertices $i \notin Q$, $\text{Dist}[i]$ is the length of the shortest path (i.e., they will not be further considered; therefore weights must not be negative!)
Dijkstra's Algorithm – Implementation (2)

Dijkstra(W: Array[1..n,1..n], r:Node) {
    ! initialise data structures
    Array Parent[1..n];
    Array Dist[1..n];
    for i from 1 to n do
        Dist[i] = inf;
    end do;
    ! init Parent and Dist for root r:
    Parent[r] = 0;
    Dist[r] = 0;
    ! init sets of explored and unexplored vertices
    Set S = {};
    Set Q = {1, .., n};
    ! ... to be continued ...

Dijkstra’s Algorithm – Implementation (3)

main loop of Dijkstra (…)

while Q <> {} do

! remove node with smallest Dist[] from Q
X = removeSmallest(Q, Dist);
S = union(S,X);
! X is added to S, thus update Dist:
forall (X,V) in X.edges do
    if V in S then continue;
    ! update neighbours of X that are not in S:
    d := Dist[X.key] + W[X.key,V.key];
    if d < Dist[V.key] then
        Dist[V.key] := d;
        Parent[V.key] := X.key;
    end if
end do;
end while;
Dijkstra’s Algorithm – Comments

• Why do we not update Dist[X.key] and Parent[X.key]?

→ this was already set in the previous iteration of the while-loop

→ how do we obtain the shortest path?

→ via the Parent[] array:

shortestPath (key: Int): List {
if Parent[key] = 0 then return [key]
else return append(shortestPath(Parent[key]), key);
end if;
}
Dijkstra’s Algorithm – Comments

• Why do we not update Dist[X.key] and Parent[X.key]?
  → this was already set in the previous iteration of the while-loop
Dijkstra’s Algorithm – Comments

- Why do we not update $\text{Dist}[X\text{.key}]$ and $\text{Parent}[X\text{.key}]$?
  → this was already set in the previous iteration of the while-loop

- how do we obtain the shortest path?
Dijkstra’s Algorithm – Comments

• Why do we not update Dist[X.key] and Parent[X.key]?
  → this was already set in the previous iteration of the while-loop

• how do we obtain the shortest path?
  → via the Parent[] array:

```c
shortestPath(key: Int) : List {
    if Parent[key] = 0
    then return [key]
    else return append(shortestPath(Parent[key]), key);
    end if;
}
```
Dijkstra’s Algorithm – Complexity

Priority Queues:

- How is the function removeSmallest implemented?

```plaintext
if d < Dist[V.key] then
    Parent[V.key] := X.key;
    Dist[V.key] := d;
    updateSorting(Q, Dist, V);
end if
```
Dijkstra’s Algorithm – Complexity

Priority Queues:

- How is the function removeSmallest implemented?
- Idea: sort elements of $Q$ according to array $Dist$
- ToDo: Update sorting of $Q$ after changes to $Dist$

```python
if d < Dist[V.key] then
    Parent[V.key] := X.key;
    Dist[V.key] := d;
    updateSorting(Q, Dist, V);
end if
```

- integrated data structure for such purposes: priority queue
Dijkstra’s Algorithm – Complexity

Priority Queues:

• How is the function removeSmallest implemented?
• Idea: sort elements of \( Q \) according to array \( \text{Dist} \)
• ToDo: Update sorting of \( Q \) after changes to \( \text{Dist} \)

```java
if d < Dist[V.key] then
    Parent[V.key] := X.key ;
    Dist[V.key] := d ;
    updateSorting(Q, Dist, V);
end if
```

• integrated data structure for such purposes: priority queue

Complexity of Dijkstra’s Algorithm:

• a complexity of \( \Theta(|E| + |V| \log |V|) \) is possible
• for dense graphs, \( |E| \in \Theta|V|^2 \), the complexity is thus \( \Theta(|V|^2) \)
Dijkstra – Single Source, Single Destination

Single Source, All Destinations:

- we can terminate Dijkstra’s Algorithm after the destination node has been removed from Q:

  \[ X = \text{removeSmallest}(Q, \text{Dist}); \]
  \[ \text{if } X = \text{destination} \text{ then return } X; \]

- otherwise Dijkstra’s Algorithm finds the shortest path from the source to all nodes in the graph.

Question:
Can Dijkstra’s Algorithm be improved, if the shortest path to only one specific destination is wanted?
Dijkstra – Single Source, Single Destination

Single Source, All Destinations:

• we can terminate Dijkstra’s Algorithm after the destination node has been removed from Q:

\[ X = \text{removeSmallest}(Q, \text{Dist}); \]
\[ \text{if } X = \text{destination} \text{ then return } X; \]

• otherwise Dijkstra’s Algorithm finds the shortest path from the source to all nodes in the graph.

Question:
Can Dijkstra’s Algorithm be improved, if the shortest path to only one specific destination is wanted?

• or more general: is there a better algorithm to solve the single-source-single-destination problem?
Dijkstra – Single Source, Single Destination

Single Source, All Destinations:

• we can terminate Dijkstra’s Algorithm after the destination node has been removed from Q:

\[
X = \text{removeSmallest}(Q, \text{Dist})
\]

\[
\text{if } X = \text{destination} \text{ then return } X;
\]

• otherwise Dijkstra’s Algorithm finds the shortest path from the source to all nodes in the graph.

Question:
Can Dijkstra’s Algorithm be improved, if the shortest path to only one specific destination is wanted?

• or more general: is there a better algorithm to solve the single-source-single-destination problem?

→ there is no algorithm known that is asymptotically faster
Minimum Spanning Tree

Definition (Minimum Spanning Tree)

A spanning tree \( T = (V, E) \) is called a **minimum spanning tree** for the graph \( G = (V, E') \), if the sum of the weights of all edges of \( T \) is minimal (among all possible spanning trees).
Minimum Spanning Tree

**Definition (Minimum Spanning Tree)**

A spanning tree \( T = (V, E) \) is called a *minimum spanning tree* for the graph \( G = (V, E') \), if the sum of the weights of all edges of \( T \) is minimal (among all possible spanning trees).

**Towards an Algorithm:**

• Dijkstra's Algorithm computes a spanning tree of shortest paths

• Idea: modify Dijkstra's "greedy approach"

→ successively add edges to a subtree

• minimise total weight of edges instead of path lengths

→ add node that is closest to the current subtree

⇒ Prim's Algorithm
Minimum Spanning Tree

Definition (Minimum Spanning Tree)

A spanning tree \( T = (V, E) \) is called a minimum spanning tree for the graph \( G = (V, E') \), if the sum of the weights of all edges of \( T \) is minimal (among all possible spanning trees).

Towards an Algorithm:

- Dijkstra’s Algorithm computes a spanning tree of shortest paths
**Minimum Spanning Tree**

**Definition (Minimum Spanning Tree)**

A spanning tree \( T = (V, E) \) is called a **minimum spanning tree** for the graph \( G = (V, E') \), if the sum of the weights of all edges of \( T \) is minimal (among all possible spanning trees).

**Towards an Algorithm:**

- Dijkstra’s Algorithm computes a spanning tree of shortest paths
- Idea: modify Dijkstra’s “greedy approach”
  \( \rightarrow \) successively add edges to a subtree
- minimise total weight of edges instead of path lengths
  \( \rightarrow \) add node that is closest to the current subtree
Minimum Spanning Tree

Definition (Minimum Spanning Tree)

A spanning tree $T = (V, E)$ is called a **minimum spanning tree** for the graph $G = (V, E')$, if the sum of the weights of all edges of $T$ is minimal (among all possible spanning trees).

Towards an Algorithm:

- Dijkstra’s Algorithm computes a spanning tree of shortest paths
- Idea: modify Dijkstra’s “greedy approach”
  → successively add edges to a subtree
- minimise total weight of edges instead of path lengths
  → add node that is closest to the current subtree

⇒ **Prim’s Algorithm**
Minimum Spanning Tree – Prim’s Algorithm

Prim (W: Array [1..n, 1..n], r:Node) {
  ! initialise data structures
  Array Parent[1..n];
  Array Nearest[1..n]; ! replaces Dist
  for i from 1 to n do
    Nearest[i] = inf;
  end do;
  ! init Parent and Dist for root r:
  Parent[r] = 0;
  Nearest[r] = 0;
  ! init sets of explored and unexplored vertices
  Set S = {};
  Set Q = {1, .., n};
  ! ... to be continued ...
Minimum Spanning Tree – Prim’s Algorithm (2)

! main loop of Prim(...)
while Q <> {} do
  ! remove nearest node from Q
  X = removeNearest(Q, Nearest);
  S = union(S, X);
  ! X is added to S, thus update Nearest:
  forall (X,V) in X.edges do
    if V in S then continue;
    ! update neighbours of X that are not in S:
    if W[X.key,V.key] < Nearest[V.key] then
      Nearest[V.key] := W[X.key,V.key];
      Parent[V.key] := X.key;
    end if
  end do;
end while;
Minimum Spanning Tree – Kruskal’s Algorithms

Another “Greedy” Algorithm:
- Idea: successively select edges with lowest weight
- but avoid cycles
- requires \texttt{union-find} data structure

\texttt{Kruskal(V,E): Set \{ \\
S := \emptyset; \\
\textbf{for all} \ v \ \textbf{in} \ V \ \textbf{do} \\
\quad \textsc{make} \textsc{-set}(v); \\
\textbf{end do}; \\
\textbf{for all} \ (u,v) \ \textbf{in} \ E \ \textbf{ordered by increasing weight}(u,v) \ \textbf{do} \\
\quad \textbf{if} \ \textsc{find} \textsc{-set}(u) \neq \textsc{find} \textsc{-set}(v) \ \textbf{then} \\
\qquad S := S \cup \{(u,v)\}; \\
\qquad \textsc{union}(u,v); \\
\textbf{end if}; \\
\textbf{end do}; \\
\textbf{return} \ S; \\
\}}
Minimum Spanning Tree

History:

- Kruskal’s algorithm: Joseph Kruskal 1956
Minimum Spanning Tree

History:

- Kruskal’s algorithm: Joseph Kruskal 1956
- Borůvka’s/Sollin’s algorithm: Otakar Borůvka 1926 (as a method of constructing an efficient electricity network for Moravia), rediscovered by Choquet 1938, Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki 1951, Sollin 1965

similar to Kruskal’s algorithm
**Minimum Spanning Tree**

**History:**

- Kruskal’s algorithm: Joseph Kruskal 1956
- Borůvka’s/Sollin’s algorithm: Otakar Borůvka 1926 (as a method of constructing an efficient electricity network for Moravia), rediscovered by Choquet 1938, Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki 1951, Sollin 1965
Minimum Spanning Tree

History:

- Kruskal’s algorithm: Joseph Kruskal 1956
- Borůvka’s/Sollin’s algorithm: Otakar Borůvka 1926 (as a method of constructing an efficient electricity network for Moravia), rediscovered by Choquet 1938, Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki 1951, Sollin 1965
  - similar to Kruskal’s algorithm