WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

1. Minimum Spanning Trees

Definition 1

Tree is a connected (path between any two nodes exists), undirected graph without cycles.

How to find possible cycles and verify whether a graph is a tree ?

1. Minimum Spanning Trees

Definition 1

Tree is a connected (path between any two nodes exists), undirected graph without cycles.

How to find possible cycles and verify whether a graph is a tree ?

Computational Problem:

Given a connected dag $G = (V, E)$ and a weight function $c : E \to \mathbf{N}$. Find a tree (V, T) that connects all nodes such that $\sum_{e \in E} c(v) \to min$.

Definition 2

A set $C \subset E$ is a *cut* if $G = (V, E - C)$ is not connected.

For $S \subset V$, $\{\{u, v\} | u \in S \wedge v \in V - S\}$ forms a cut.

Computational Problem:

Given a connected dag $G = (V, E)$ and a weight function $c : E \to \mathbf{N}$. Find a tree (V, T) that connects all nodes such that $\sum_{e \in E} c(v) \to min$.

Definition 2

A set $C \subset E$ is a *cut* if $G = (V, E - C)$ is not connected.

For $S \subset V$, $\{\{u, v\} | u \in S \land v \in V - S\}$ forms a cut.

Theorem 3

A lightest edge in a cut can be used in an MST.

Proof.

Suppose MST T' uses edge e' between S and $V - S$ and $c(e) \leq c(e')$. Then $T = T' - \{e'\} \cup \{e\}$ is also an MST. □

Theorem 3

A lightest edge in a cut can be used in an MST.

Proof.

Suppose MST T' uses edge e' between S and $V - S$ and $c(e) \leq c(e')$. Then $T = T' - \{e'\} \cup \{e\}$ is also an MST. □

Theorem 4

A heaviest edge on a cycle is not needed for an MST.

Proof.

Suppose MST T' uses heaviest edge e' on cycle C and $c(e) \leq c(e')$. Then $T = T' - \{e'\} \cup \{e\}$ is also an MST. $\square$

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

1.1 Algorithm of Prim

- Input: A connected weighted graph $G = (V, E)$
- Initialize: $V_{new} = \{x\}$, where x is an arbitrary node, $E_{new} = \{\}$
- Repeat until $V_{new} = V$:
  - Choose edge $\{u, v\}$ from E with minimal weight such that $u \in V_{new}$ and v not.
  - Add v to V_{new} and $\{u, v\}$ to E_{new}
- Output: V_{new}, E_{new}.

What is about complexity of this algorithm ? Obviously, the complexity depends on the way how the graph is stored.

- adjacency matrix: $O(|V|^2)$
- using *Priority Queues* based on Fibonacci-Heaps: $O(|E| + |V|log(|V|))$

What is about complexity of this algorithm ? Obviously, the complexity depends on the way how the graph is stored.

- adjacency matrix: $O(|V|^2)$
- using *Priority Queues* based on Fibonacci-Heaps: $O(|E| + |V| log(|V|))$

What is about complexity of this algorithm ? Obviously, the complexity depends on the way how the graph is stored.

- adjacency matrix: $O(|V|^2)$
- using *Priority Queues* based on Fibonacci-Heaps: $O(|E| + |V|log(|V|))$

What is about complexity of this algorithm ? Obviously, the complexity depends on the way how the graph is stored.

- adjacency matrix: $O(|V|^2)$
- using *Priority Queues* based on Fibonacci-Heaps: $O(|E| + |V| log(|V|))$

1.2 Prim algorithm using Priority Queues

Priority Queue is a data structure supporting the following operations:

- *insert_element* - $O(log(|E| + |V|))$
- *delete_min* - $O(|E|log(|E| + |V|))$
- *decrease_key* - $O(|E|log(|E| + |V|))$