Algorithms and Theory of Computation

Lecture 5: Minimum Spanning Tree

Xiaohui Bei

MAS 714

August 27, 2018
Interval Scheduling

**Input:** a set of jobs, job $j$ start at $s_j$ and finishes at $f_j$

Two jobs are **compatible** if they don’t overlap.

**Goal:** find a maximum subset of mutually compatible jobs.
Algorithm: SomeGreedyAlgorithm(G):

Let $R$ be the set of all jobs.;
Initialize $X = \emptyset$; // $X$ will store all scheduled jobs

while $R$ is not empty do

choose $j \in R$ with the smallest $f_j$;
add $j$ to $X$;
remove from $R$ all jobs that overlap with $j$;

return $X$

Theorem

The greedy algorithm that picks jobs in the ascending order of their finishing times is optimal.
Proving Optimality

Feasibility: Clearly the algorithm returns a set of jobs that are mutually compatible.

Optimality: Given job set $R$, let $O$ be an optimal set and let $X$ be the set returned by the greedy algorithm.

▶ Can we show $X = O$?

Not likely!

▶ Instead we will show that $|O| = |X|$. 

Nanyang Technological University
MAS 714
August 27, 2018
Feasibility: Clearly the algorithm returns a set of jobs that are mutually compatible.
Proving Optimality

- **Feasibility:** Clearly the algorithm returns a set of jobs that are mutually compatible.

- **Optimality:** Given job set $R$, let $O$ be an optimal set and let $X$ be the set returned by the greedy algorithm.
  - Can we show $X = O$?
Proving Optimality

- **Feasibility:** Clearly the algorithm returns a set of jobs that are mutually compatible.

- **Optimality:** Given job set $R$, let $O$ be an optimal set and let $X$ be the set returned by the greedy algorithm.
  - Can we show $X = O$? Not likely!
Proving Optimality

- **Feasibility:** Clearly the algorithm returns a set of jobs that are mutually compatible.

- **Optimality:** Given job set $R$, let $O$ be an optimal set and let $X$ be the set returned by the greedy algorithm.
  - Can we show $X = O$? Not likely!
  - Instead we will show that $|O| = |X|$
Proof idea: convert an optimal solution $O$ to the greedy solution $X$ while maintaining its cardinality.

Proof.

1. Let $x_1, x_2, \ldots, x_k$ denote set of jobs selected by greedy.
2. Let $o_1, o_2, \ldots, o_m$ denote set of jobs in an optimal solution.
Proof idea: convert an optimal solution $O$ to the greedy solution $X$ while maintaining its cardinality.

Proof.

1. Let $x_1, x_2, \ldots, x_k$ denote set of jobs selected by greedy.
2. Let $o_1, o_2, \ldots, o_m$ denote set of jobs in an optimal solution.
3. For each $r$ from 1 to $k$:
   - In the optimal solution, replace job $o_r$ with job $x_r$.
     - job $x_r$ exists and finishes before $o_r$
     - new solution still feasible and optimal
4. In the end, the optimal solution will coincide with the greedy solution.
Exercise: shortest interval algorithm always picks at least half the optimum number of jobs.

How about when jobs arrives in an online fashion?
  ▶ An active research subject.
More about Interval Scheduling

1. Exercise: shortest interval algorithm always picks at least half the optimum number of jobs.

2. How about when jobs arrives in an online fashion?
   - An active research subject.

3. Weighted Interval Scheduling: each job \( j \) also has a weight \( w_i \), want to find a set of compatible jobs whose total weight is maximized.
   - (A) Earliest start time first.
   - (B) Earliest finish time first.
   - (C) Highest weight first.
   - (D) None of the above.
   - (E) IDK
More about Interval Scheduling

1 Exercise: shortest interval algorithm always picks at least half the optimum number of jobs.

2 How about when jobs arrives in an online fashion?
   ▶ An active research subject.

3 Weighted Interval Scheduling: each job $j$ also has a weight $w_i$, want to find a set of compatible jobs whose total weight is maximized.
   (A) Earliest start time first.
   (B) Earliest finish time first.
   (C) Highest weight first.
   (D) None of the above.
   (E) IDK
A weighted graph is a graph in which each edge $e$ is associated with a numerical weight $c_e$. 
Minimum Spanning Trees (MST)

A weighted graph is a graph in which each edge $e$ is associated with a numerical weight $c_e$.

Spanning Tree

A spanning tree of an undirected connected graph $G = (V, E)$ is a set of edges $T \subseteq E$ such that

- $T$ forms a tree, and
- $(V, T)$ is connected.

In a weighted graph $G$, a minimum spanning tree is a spanning tree with the smallest sum of edge weights.
Minimum Spanning Trees (MST)

A weighted graph is a graph in which each edge $e$ is associated with a numerical weight $c_e$.

Spanning Tree

A spanning tree of an undirected connected graph $G = (V, E)$ is a set of edges $T \subseteq E$ such that

- $T$ forms a tree, and
- $(V, T)$ is connected.

In a weighted graph $G$, a minimum spanning tree is a spanning tree with the smallest sum of edge weights.

MST

Find a minimum spanning tree of a weighted graph $G$. 
Example

Diagram of a network with nodes labeled a through g, connected by weighted edges.

- Edge between a and b labeled 7
- Edge between a and d labeled 5
- Edge between b and c labeled 8
- Edge between b and e labeled 7
- Edge between b and f labeled 6
- Edge between b and g labeled 11
- Edge between d and e labeled 15
- Edge between d and f labeled 9
- Edge between d and g labeled 8
- Edge between e and f labeled 5
- Edge between e and g labeled 9

Network structure and weights provided for analysis.
Example
Applications

A basic and fundamental problem in graph theory, often used to measure costs to establish connections between vertices.

1. **Network Design:** designing networks with minimum cost while guaranting connectivity

2. **Approximation Algorithm:** can be used to approximate other hard problems such as Traveling Salesman Problem, Steiner Trees, etc.
Algorithm: SomeGreedyMSTAlgorithm(G):

Initialize $T = \emptyset$ ; // $T$ will store edges of a MST

while $T$ is not a spanning tree of $G$ do

    choose $e \in E$ that satisfies condition;

    add $e$ to $T$;

return $T$
Greedy Template

Algorithm: \texttt{SomeGreedyMSTAlgorithm}(G):
- Initialize $T = \emptyset$;  // $T$ will store edges of a MST
- \textbf{while} $T$ is not a spanning tree of $G$ \textbf{do}
  - choose $e \in E$ that satisfies condition;
  - add $e$ to $T$;
- \textbf{return} $T$

Which edges, and in what order, should be processed and added to the spanning tree?
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to \( T \) as long as they don’t form a cycle.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

![Graph](image-url)
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.
Prim’s Algorithm

$T$ maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to $T$. 

![Graph](image)
Prim’s Algorithm

T maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachment cost to T.
Prim’s Algorithm

$T$ maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachment cost to $T$. 

![Graph Diagram]

Nanyang Technological University

MAS 714

August 27, 2018

12 / 23
Prim’s Algorithm

$T$ maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to $T$. 
Prim’s Algorithm

$T$ maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to $T$. 

![Graph Diagram]
Prim’s Algorithm

The graph maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to $T$. 

![Graph Diagram]

Nanyang Technological University  
MAS 714  
August 27, 2018  12 / 23
Prim’s Algorithm

$T$ maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to $T$. 
Prim’s Algorithm

T maintained by the algorithm will be a tree, starting from a single vertex. In each iteration, pick edges with least attachedment cost to T.
Correctness of MST algorithms

- Many different MST algorithms.
- All of them rely on some basic properties of MSTs.

Assumption

For simplicity, we assume that edge costs are distinct, that is no two edge costs are equal.
Cut

Given a graph $G = (V, E)$, a cut is a partition of the vertices into two disjoint subsets. Edges that have one endpoint in each subset of the partition are the edges of the cut, and these edges are said to cross the cut.
Given a graph $G = (V, E)$, a cut is a partition of the vertices into two disjoint subsets. Edges that have one endpoint in each subset of the partition are the edges of the cut, and these edges are said to cross the cut.
Safe Edges

An edge $e$ is a **safe** edge if there exists a partition of $V$ into $S$ and $V \setminus S$, and $e$ is the unique minimum cost edge crossing this partition.
Safe Edges

An edge $e$ is a safe edge if there exists a partition of $V$ into $S$ and $V \setminus S$, and $e$ is the unique minimum cost edge crossing this partition.

Safe edge in the cut $(S, V \setminus S)$
Theorem

Given an undirected connected graph $G$ with distinct edge costs, the set of safe edges in $G$ form the unique MST of $G$. 

Proof.

Prove the lemma in two steps:

1. If $e$ is a safe edge, then every MST contains $e$.
2. The set of safe edges form a connected graph that covers every vertex.
**Theorem**

Given an undirected connected graph $G$ with distinct edge costs, the set of safe edges in $G$ form the unique MST of $G$.

**Proof.**

Prove the lemma in two steps:

1. If $e$ is a safe edge, then every MST contains $e$.
2. The set of safe edges form a connected graph that covers every vertex.
Lemma 1

If $e$ is a safe edge, then every MST contains $e$. 

Proof.
Assume by contradiction that $e$ is not in some MST $T$.

1. $e = \{u, v\}$ is safe $\Rightarrow$ there is an $S \subset V$ such that $e$ is the unique minimum cost edge crossing $S$.

2. Adding $e$ to $T$ creates a cycle $C$ in $T \cup \{e\}$.

3. There must exist another edge $e'$ in $T$ crossing this cut.

4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is a spanning tree of lower cost.
Cut Property

**Lemma 1**
If \( e \) is a safe edge, then every MST contains \( e \).

**Proof.**
Assume by contradiction that \( e \) is not in some MST \( T \).

1. \( e = \{u, v\} \) is safe \( \implies \) there is an \( S \subset V \) such that \( e \) is the unique minimum cost edge crossing \( S \).
2. Adding \( e \) to \( T \) creates a cycle \( C \) in \( T \cup \{e\} \).
3. There must exist another edge \( e' \) in \( T \) cross this cut.
4. \( T' = (T \setminus \{e'\}) \cup \{e\} \) is a spanning tree of lower cost.
Lemma 2
The set of safe edges from a connected graph that covers every vertex.
Lemma 2

The set of safe edges from a connected graph that covers every vertex.

Proof.

1. Assume not. Let \( S \) be a connected component in the graph induced by safe edges.

2. Let \( e \) be the smallest cost edge crossing \( S \) \( \Rightarrow \) \( e \) is a safe edge.
Kruskal’s Algorithm

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

**Algorithm: Kruskal($G$):**

1. Initialize $T = \emptyset$;  // $T$ will store edges of a MST
2. **while** $T$ is not a spanning tree of $G$ **do**
   1. choose $e \in E$ of minimum cost;
   2. remove $e$ from $E$;
   3. **if** $T \cup \{e\}$ does not contain cycles **then**
      1. add $e$ to $T$;
3. return $T$
Correctness

Process edges in the order of their costs (in increasing order), and add edges to \( T \) as long as they don’t form a cycle.
Correctness

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

Only need to show that all edges added are safe.

Proof of Correctness.

- When $e = (u, v)$ is added to the tree, let $S$ and $S'$ be the connected components containing $u$ and $v$ respectively.
- $e$ has minimum cost under all edges forming no cycle, hence $e$ has minimum cost among all edges crossing the cut $S$ (and also $S'$)
- $e$ is a safe edge.
Time Complexity Analysis

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

**Algorithm:** Kruskal($G$):

Initialize $T = \emptyset$ ;  // $T$ will store edges of a MST
while $T$ is not a spanning tree of $G$ do
    choose $e \in E$ of minimum cost;
    remove $e$ from $E$;
    if $T \cup \{e\}$ does not contain cycles then
        add $e$ to $T$;
    return $T$
Time Complexity Analysis

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

Algorithm: $\text{Kruskal}(G)$:

- Initialize $T = \emptyset$ ;  // $T$ will store edges of a MST
- **while** $T$ is not a spanning tree of $G$ **do**
  - choose $e \in E$ of minimum cost;
  - remove $e$ from $E$;
  - **if** $T \cup \{e\}$ does not contain cycles **then**
    - add $e$ to $T$;
- return $T$

- Presort edges by costs. Choosing minimum then takes $O(1)$ time.
Time Complexity Analysis

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

Algorithm: **Kruskal**($G$):

1. Initialize $T = \emptyset$; // $T$ will store edges of a MST
2. while $T$ is not a spanning tree of $G$ do
   1. choose $e \in E$ of minimum cost;
   2. remove $e$ from $E$;
   3. if $T \cup \{e\}$ does not contain cycles then
      1. add $e$ to $T$;
   4. return $T$

- Presort edges by costs. Choosing minimum then takes $O(1)$ time.
- Do DFS on $T \cup \{e\}$. Takes $O(n)$ time.
Time Complexity Analysis

Process edges in the order of their costs (in increasing order), and add edges to $T$ as long as they don’t form a cycle.

**Algorithm: Kruskal($G$):**

Initialize $T = \emptyset$;  // $T$ will store edges of a MST

while $T$ is not a spanning tree of $G$ do
   choose $e \in E$ of minimum cost;
   remove $e$ from $E$;
   if $T \cup \{e\}$ does not contain cycles then
      add $e$ to $T$;

return $T$

- Presort edges by costs. Choosing minimum then takes $O(1)$ time.
- Do DFS on $T \cup \{e\}$. Takes $O(n)$ time.
- Total time $O(m \log m) + O(mn) = O(mn)$. 

More Efficient Implementation

Algorithm: SmarterKruskal\((G)\):

1. Initialize \(T = \emptyset\); // \(T\) will store edges of a MST
2. Put each vertex \(u \in V\) into a set by itself;
3. \(\text{foreach} e = \{u, v\} \in E\) in the order of increasing costs \(\text{do}\)
   1. \(\text{if} u\) and \(v\) belong to different sets \(\text{then}\)
      1. \(\text{add} e\) to \(T\);
      2. \(\text{merge the two sets containing} u\) and \(v\);
4. \(\text{return} T\);

Need a data structure to:

- check if two elements belong to the same set
- merge two sets
Algorithm: SmarterKruskal(G):

1. Initialize $T = \emptyset$;  // $T$ will store edges of a MST
2. Put each vertex $u \in V$ into a set by itself;
3. foreach $e = \{u, v\} \in E$ in the order of increasing costs do
   4. if $u$ and $v$ belong to different sets then
      5. add $e$ to $T$;
      6. merge the two sets containing $u$ and $v$;
5. return $T$
More Efficient Implementation

Algorithm: \texttt{SmarterKruskal}(G):

Initialize $T = \emptyset$; // $T$ will store edges of a MST

Put each vertex $u \in V$ into a set by itself;

\textbf{foreach} $e = \{u, v\} \in E$ \textit{in the order of increasing costs} \textbf{do}

\hspace{1em} \textbf{if} $u$ and $v$ belong to different sets \textbf{then}

\hspace{2em} add $e$ to $T$;

\hspace{2em} merge the two sets containing $u$ and $v$;

\textbf{return} $T$

Need a data structure to:

- check if two elements belong to same set
- merge two sets
Data Structure: Union-Find

Union-Find

Store a set of disjoint sets with the following operations:

1. **Make-Set**($V$): generate a set $\{v\}$ for each vertex $v \in V$. Name of set $\{v\}$ is $v$.
2. **Find**($u$): find the name of the set containing vertex $u$.
3. **Union**($u$, $v$): merge the sets named $u$ and $v$. Name of the new set is either $u$ or $v$. 

The running time of Kruskal algorithm will depend on the implementation of the data structure.
Data Structure: Union-Find

Union-Find

Store a set of disjoint sets with the following operations:

1. Make-Set$(V)$: generate a set $\{v\}$ for each vertex $v \in V$. Name of set $\{v\}$ is $v$.
2. Find$(u)$: find the name of the set containing vertex $u$.
3. Union$(u, v)$: merge the sets named $u$ and $v$. Name of the new set is either $u$ or $v$.

The running time of Kruskal algorithm will depend on the implementation of the data structure.