Algorithms and Theory of Computation

Lecture 6: Minimum Spanning Tree (2)

Xiaohui Bei

MAS 714

August 28, 2018
More Efficient Implementation

Algorithm: SmarterKruskal(G):

Initialize $T = \emptyset$ ; // $T$ will store edges of a MST
Put each vertex $u \in V$ into a set by itself;

foreach $e = \{u, v\} \in E$ in the order of increasing costs do
    if $u$ and $v$ belong to different sets then
        add $e$ to $T$;
        merge the two sets containing $u$ and $v$;
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.
Union-Find: Implementation

Sets are represented as trees, by pointers towards the roots. All elements in one tree belong to a set with root’s name.

- Find\((u)\): Traverse from \(u\) to the root
- Union\((u, v)\): Make root of \(u\) (smaller set) point to root of \(v\). Takes \(O(1)\) time.

Each vertex \(u\) has a pointer \(\text{parent}(u)\) to its ancestor.

![Figure](image.png)

Figure: Union(Find\((v)\), Find\((u)\))
Algorithm: \textbf{Make-Set}(G):

\begin{verbatim}
foreach \( u \in V \) do
    parent(u) = u;
\end{verbatim}

Algorithm: \textbf{Find}(u):

\begin{verbatim}
while parent(u) \neq u do
    u = parent(u);
return u
\end{verbatim}

Algorithm: \textbf{Union}(u, v):

\begin{verbatim}
(* parent(u) = u & parent(v) = v *)
if \(|\text{component}(u)| \leq |\text{component}(v)|\) then
    parent(u) = v
else
    parent(v) = u
set new component size to \(|\text{component}(u) + \text{component}(v)|\).
\end{verbatim}
Analysis

- Make-Set: $O(n)$ time.
- Union: $O(1)$ time.
- Find: $O(\text{depth of the tree})$ time.

Proposition

The maximum depth of trees in union-find is $O(\log n)$.

Proof.

- Depth of tree($u$) increases by at most 1 only when the set containing $u$ changes its name.
- If depth of tree $u$ increases then the size of the set containing $u$ (at least) doubles.
- Maximum set size is $n$; so the depth of any tree is at most $O(\log n)$. 
Speed up!

- When calling $\text{Find}(u)$, we traverse the path from $u$ to the root.
- Consecutive calls of $\text{find}(u)$ traverse the same path.

Idea: Path Compression
Make all vertices on the path in $\text{Find}(u)$ point to root directly.
Path Compression: Example

Algorithm: Find(u):

```
if parent(u) \neq u then
    parent(u) = Find(parent(u));
return parent(u)
```

![Figure](image1.png)

![Figure: After Find(u)](image2.png)
Path Compression

**Question**
Does Path Compression help?

Yes!

**Theorem**
With Path Compression, the amortized running time of \textbf{Find} operations is $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the \textbf{Ackermann function} $A(n, n)$. 
Ackermann and Inverse Ackermann Functions

Ackermann function $A(m, n)$ defined for $m, n \geq 0$:

$$A(m, n) = \begin{cases} 
  n + 1 & \text{if } m = 0 \\
  A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
  A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0 
\end{cases}$$

- $A(3, n) = 2^{n+3} - 3$
- $A(4, 3) = 2^{2^{65536}} - 3$

$\alpha(n)$ is the inverse of $A(n, n)$

For all practical purposes, $\alpha(n) \leq 5$. 
Running time of Kruskal’s Algorithm

Using Union-Find data structure, Kruskal’s Algorithm takes
- \( O(m) \) \textit{Find} operations (two for each edge)
- \( O(n) \) \textit{Union} operations (one for each edge added to \( T \))
- 1 sorting operation

Total time = \( O(m\alpha(n) + n + m \log m) = O(m \log m) \)
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 $T$.

**Algorithm: Prim($u$):**

1. Initialize $T = \emptyset$ ;  // $T$ will store edges of a MST
2. Initialize $S = \{1\}$;
3. while $T$ is not a spanning tree of $G$ do
   1. choose $e = (u, v) \in E$ of minimum cost such that $u \in S$ and $v \in V - S$;
   2. $T = T \cup \{e\}$;
   3. $S = S \cup \{v\}$;
4. return $T$
Correctness

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

Proof of correctness.

1. If $e$ is added to the tree, then $e$ is safe
   - Let $S$ be the vertices connected by edges in $T$ when $e$ is added.
   - $e$ is the minimum cost edge crossing cut $(S, V\setminus S)$.

2. $S$ is connected in each iteration and eventually $S = V$. 
Time Complexity Analysis

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

Algorithm: $\text{Prim}(u)$:

- Initialize $T = \emptyset$ ; // $T$ will store edges of a MST
- Initialize $S = \{1\}$;
- \hspace{1cm} while $T$ is not a spanning tree of $G$ do
- \hspace{2cm} choose $e = (u, v) \in E$ of minimum cost
- \hspace{2cm} such that $u \in S$ and $v \in V - S$;
- \hspace{2cm} $T = T \cup \{e\}$;
- \hspace{2cm} $S = S \cup \{v\}$;
- \hspace{1cm} return $T$

- $O(n)$ iterations
- $O(m)$ time to pick edge $e$ in each iteration
- Total running time $= O(mn)$
Algorithm: SmarterPrim(u):

Initialize $T = \emptyset$ ; $\quad$ // $T$ will store edges of a MST
Initialize $S = \{1\}$;
for $u \notin S$, $a(u) = \arg \min_{e=(u,v), v \in S} c_e$;
while $T$ is not a spanning tree of $G$ do
$\quad$ pick minimum $a(u) = (u, v)$ ;
$\quad$ $T = T \cup \{a(u)\}$;
$\quad$ $S = S \cup \{u\}$;
$\quad$ update array $a$;
return $T$

Maintain vertices in $V \setminus S$ in a priority queue.
# Priority Queue

## Priority Queues

Store a set $S$ of $n$ elements, where each element $v \in S$ has an associated real/integer key $k(v)$, with the following operations:

1. **Make-Queue**: create an empty queue
2. **Find-Min**: find the minimum key in $S$
3. **Extract-Min**: remove $v \in S$ with the smallest key and return it
4. **Decrease-Key** $(v, k'(v))$: decrease key of $v$ from $k(v)$ to $k'(v)$
5. **Add** $(v, k(v))$: add new element $v$ with key $k(v)$ to $S$

Very useful data structure, will discuss in detail in later lectures.

Prim requires $O(n)$ Extract-Min and $O(m)$ Decrease-Key operations.

- Using standard Heaps, total time $= O((m + n) \log n)$.
- Using Fibonacci Heaps, total time $= O(n \log n + m)$. 
More about MST

- There is an algorithm that runs in $O(n + m\alpha(n))$ time.
- There is a randomized algorithm that runs in $O(m + n)$ expected time.
- There is an algorithm using bit operations in RAM model that runs in $O(m + n)$ time.
- Still open: Is there an $O(m + n)$ time deterministic algorithm in the comparison model?