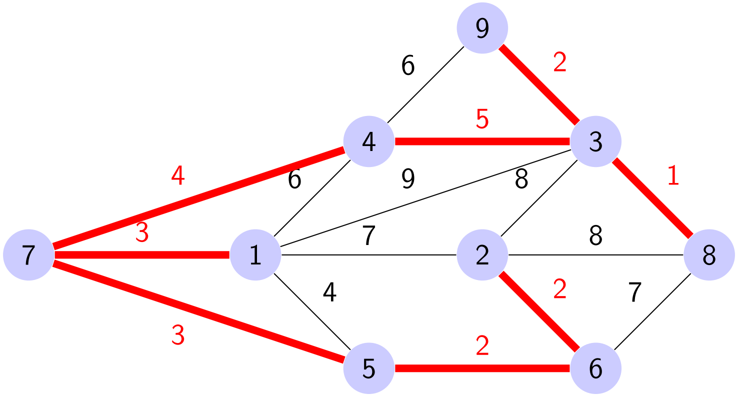
CSE 431/531: Analysis of Algorithms (Summer 2023)
Minimum Spanning Tree

Chen Xu

July 20, 2023

Minimum Spanning Tree

- A minimum spanning tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph.
- It connects all the vertices together, without any cycles and with the minimum possible total edge weight.



Two algorithms of finding the MST

- There are two greedy algorithms that find the MST of a given weighted undirected graph.
  1. Kruskal's algorithm
  2. Prim's algorithm
- They both are choosing the edges one by one and pretty efficient.
- The strategies of choosing the edges and the implementations are quite different.

Kruskal's algorithm

- Kruskal's algorithm chooses one **possible** edge with the minimum weight every step. As long as this edge does not form a cycle with the previous edges we choose it. After all edges form one single connected component for the whole graph, the algorithm finishes.
- We will use **Union-Find** data structure to maintain the connected components.

Union Find Data Structure

- The Union-Find data structure is used to manage partitions of a set into disjoint subsets.
- The two main operations are:
  - **Find:** Determine which subset a particular element is in.
  - **Union:** Merge two subsets into a single subset.

1	2	3	4	5	6	7	8	9
-2	1	-1	-3	4	-2	6	-1	4

- All elements are initialized with -1. When it is a negative number the absolute value is the size of the partition. When it is a positive number it points to the partition that it belongs to. There are techniques of **rank** and **path compression** to optimize the performance of this data structure.

Kruskal algorithm example

Disjoint Set

0	2
1	2
2	-8
3	7
4	2
5	2
6	2
7	2

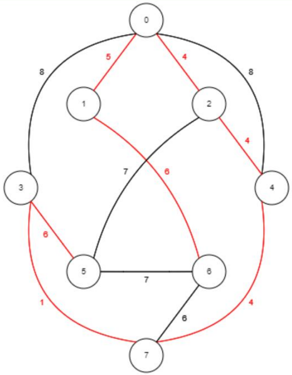
$0 \overset{8}{\text{———}} 3$

$0 \overset{8}{\text{———}} 4$

$6 \overset{6}{\text{———}} 7$

$2 \overset{7}{\text{———}} 5$

$5 \overset{7}{\text{———}} 6$



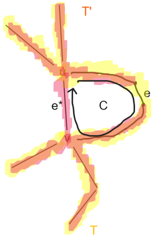
Kruskal's algorithm code using Union-Find

Kruskal MST

```
1:  KRUSKAL($V$, $E$)
2:      $A \leftarrow \emptyset$
3:      for each vertex $v \in V$ do
4:          $uf$.MAKE-SET($v$)
5:      sort the edges of $E$ into non-decreasing order by weight $w$
6:      for $(u, v) \in E$ with the min weight do
7:          if $uf$.FIND($u$) $\neq$ $uf$.FIND($v$) then
8:              $A \leftarrow A \cup \{(u, v)\}$
9:              $uf$.UNION($u, v$)
10:     return $A$
```

Correctness and running time

- Why does taking the min weight edge every step make sense? Take a minimum spanning tree T assume the lightest edge $e^* = (u, v) \notin T$ then there is another unique path in T connecting u and v. This path with e^* forms a cycle C. Remove any edge e other than e^* in C we obtain tree T', $w(e^*) \leq w(e) \to w(T') \leq w(T)$ therefore T' is also a MST.



- Running time is $O(m \log m) = O(m \log(n^2)) = O(m \log n)$.

Prim's Algorithm

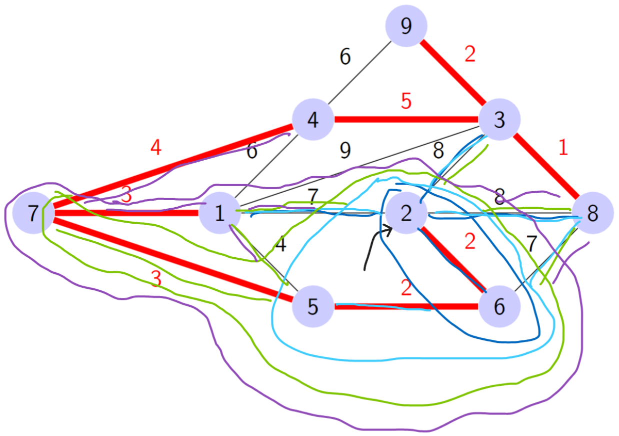
- Prim's algorithm first picks a random vertex v put it in the set S then start to include more vertices one by one into S. Every time it picks the vertex that has the min weight edge incident to S. That edge is considered in the tree too. The procedure stops when all vertices are included in S.

Simple version of Prim's algorithm

Naive Prim's algorithm

```
1: PRIM($V, E$)
2:     select an arbitrary vertex $v_0$ from $V$ to start the MST
3:     let $MST$ be the set containing $v_0$
4:     while $MST \neq V$ do
5:         select an edge $(u \in MST, v \notin MST)$ of minimum weight
6:         add $v$ to the $MST$
```

Prim's algorithm example



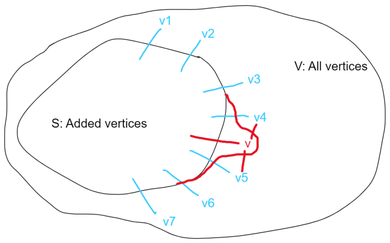
S_1
S_2
S_3
S_4
⋮

Correctness

- We can show the correctness by contradiction.
- Suppose T is an MST achieved by not picking the least weight edge during steps of Prim's algorithm. Let that step be i and let the set of the vertices already covered be W. We have an edge e_i picked instead of the least weight e_i^*. We have $w(e_i) > w(e_i^*)$. So T has e_i, let $T^* = T \setminus \{e_i\} \cup \{e_i^*\}$, it can be shown that T^* is also a spanning tree and $w(T^*) < w(T)$ this contradicts the fact that T is an MST.

Efficient implementation

- The bottleneck of the naive Prim's algorithm is that when we consider the least weight edge to be added we need to scan for all edges. There are a lot of unnecessary scans.
- Since every time we are just adding one edge and one vertex, we can maintain a list of information of how far the min distance of each vertex is to the collection of added vertices as well as what is the closest vertex in the set of added vertices.



Improved version of Prim's algorithm

We need:

- $d(v)$: The array of the least weight of every vertex v to the set S. (That is v to one of the vertex in S).
- $\pi(v)$: The array of the corresponding vertex $u \in S$ such that (u, v) has the least weight.
- Update the above arrays whenever S increases size every iteration.

Improved Prim's algorithm

Prim MST

```
1:  $s \leftarrow \textsc{ArbitraryVertex}(G)$
2:  $S \leftarrow \emptyset$
3:  $d[s] \leftarrow 0$
4:  for all $v \in V \setminus \{s\}$ do
5:      $d[v] \leftarrow \infty$
6:  while $S \neq V$ do
7:      $u \leftarrow$ a vertex in $V \setminus S$ that has minimum value in $d[*]$.
8:      $S \leftarrow S \cup \{u\}$
9:      for all $v \in V \setminus S$ such that $(u, v) \in E$ do
10:         if $w(u, v) < d[v]$ then
11:             $d[v] \leftarrow w(u, v)$
12:             $\pi[v] \leftarrow u$
13: return $\{(u, \pi[u]) \mid u \in V \setminus \{s\}\}$
```

We can maintain $d[*]$ and $\pi[*]$ by a data structure called **Priority Queue**.

Priority Queue

- A priority queue is an abstract data structure that maintains a set U of elements, each with an associated key value, and supports the following operations:
    1. $\textit{insert}(v, \textit{keyvalue})$: insert an element v, whose associated key value is key value.
    2. $\textit{decrease\_key}(v, \textit{newkeyvalue})$: decrease the key value of an element v in queue to new key value
    3. $\textit{extract\_min}()$: return and remove the element in queue with the smallest key value
- Depending on implementation. The *decrease_key* operation can be achieved in $O(\log n)$ by heap and $O(1)$ by fibonacci heap. The *extract_min* can be achieved in $O(\log n)$.

Prim's algorithm with Priority Queue

Prim MST

```
1: $s \leftarrow$ arbitrary vertex in G
2: $S \leftarrow \emptyset$, $d(s) \leftarrow 0$ and $d[v] \leftarrow \infty$ for every $v \in V \setminus \{s\}$
3: $Q \leftarrow$ empty queue, for each $v \in V$ : $Q$.insert($v$, $d[v]$)
4: while $S \neq V$ do
5:     $u \leftarrow Q$.extract_min()
6:     $S \leftarrow S \cup \{u\}$
7:     for each $v \in V \setminus S$ such that $(u, v) \in E$ do
8:         if $w(u, v) < d[v]$ then
9:             $d[v] \leftarrow w(u, v)$, $Q$.decrease_key($v$, $d[v]$)
10:            $\pi[v] \leftarrow u$
11: return $\{(u, \pi[u]) | u \in V \setminus \{s\}\}$
```

Running time is $O(n \log n + m)$ for fibonacci heap and $O(m \log n)$ for heap.