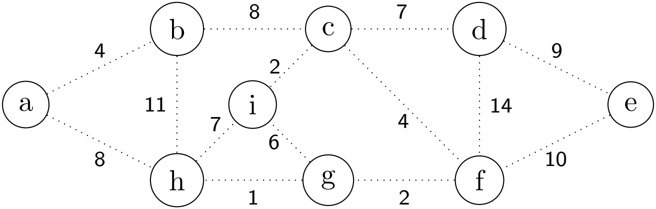
CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1]based on notes by Anna Bretscher and Albert Lai

introduction



An (edge-)weighted graph

Applications?

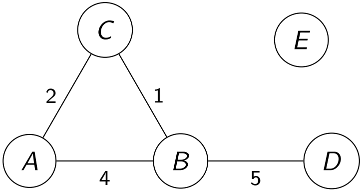
-

weighted graph

A weighted (edge-weighted) graph consists of:

- a set of vertices V
- a set of edges E
- weights: a map $w : E \to \mathbb{R}$ (usually ≥ 0)
  - if undirected graph: (u, v) and (v, u) have the same weight
  - if directed graph: (u, v) and (v, u) may have different weights

storing a weighted graph



Adjacency matrix:

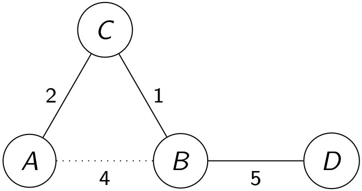
	A	B	C	D	E
A	0	4	2	∞	∞
B	4	0	1	5	∞
C	2	1	0	∞	∞
D	∞	5	∞	0	∞
E	∞	∞	∞	∞	0

Adjacency lists:

	adjacency list
A	$(B,4)$, $(C,2)$
B	$(A,4)$, $(C,1)$, $(D,5)$
C	$(A,2)$, $(B,1)$
D	$(B,5)$
E	

minimum spanning tree

- common task #1 on weighted graphs
- find a spanning tree
  - a tree that covers all vertices
  - a tree T such that every vertex $v \in V$ is an endpoint of at least one edge in T
- minimise the sum of the weights of the edges used
  - $weight(T) = \sum_{(u,v) \in T} weight(u, v)$
  - want tree T with minimum $weight(T)$



Usually just for undirected, connected graphs.

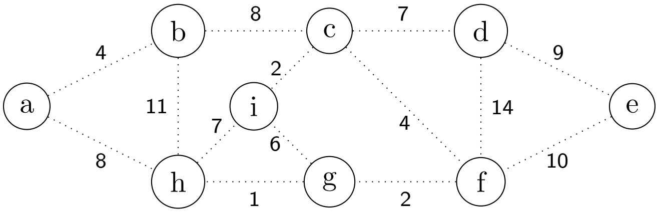
Kruskal's algorithm: idea

Kruskal's algorithm finds a MST by successive mergers.

1. At first, each vertex is its own small cluster/tree/set.
2. Find an edge of minimum weight, use it to merge two clusters/trees/sets into one.
   - Do not create cycles!
3. Do it again...
4. In general, find an edge of minimum weight that crosses two clusters; merge them into one.

Correctness idea: at each iteration find the cheapest way to merge two trees.

Kruskal's algorithm: example




```
L: [(g,h,1), (c,i,2), (f,g,2), (c,f,4), (a,b,4),
    (g,i,6), (c,d,7), (h,i,7), (a,h,8), (b,c,8),
    (d,e,9), (e,f,10), (b,h,11), (d,f,14)]
```

Clusters:

MST:

Kruskal's algorithm

```
T := new container for edges
L := edges sorted in non-decreasing order by weight
for each vertex v:
  v.cluster := make-cluster(v)
for each (u, v) in L:
  if u.cluster != v.cluster:
    T.add((u,v))
    merge u.cluster and v.cluster
return T
```

storing clusters

An easy way for now:

- each cluster is a linked list
- v.cluster is pointer to v's owning linked list
- u.cluster $\neq$ v.cluster is:
- merging two clusters is merging two linked lists:
  - a lot of vertices may need their v.cluster's updated!

storing clusters

An easy way for now, continued...

Choose to always move the smaller list to the larger one:

- in the best case:
- in the worst case:
- in the worst case:
- then how many such merges can we do?
- each v.cluster is updated at most:

A much better way will appear later in this course.

Kruskal's algorithm: time

Let $n = |V|$ and $m = |E|$. Then:

- Collecting and sorting edges:
- v.cluster updates:
- the rest is $\Theta(1)$ per vertex or edge

Total:

But lets look at n and m:

- maximum number of edges in a graph with n vertices:
- then

Then total time is

Prim's algorithm: idea

Prim's algorithm finds a MST by a BFS with a twist:

- the queue is replaced with a minimum priority queue
- with an additional operation `decrease-priority(vertex, new-priority)`
  - **Exercise**: show that `decrease-priority` is $\mathcal{O}(\log n)$ where n is the size of the priority queue

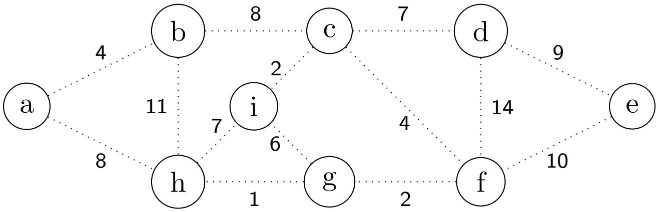
Keep unvisited vertices in the priority queue:

$$priority(v) = \text{minimum weight of any edge between } v \text{ and tree}$$
$$priority(v) = \infty \text{ if no such edge}$$

The algorithm grows a tree by one edge at a time.

Correctness idea: every time we `extract-min`, we get the cheapest edge to add to the tree.

Prim's algorithm: example



Priority queue contains vertices *not* in tree:

vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									

MST:

Prim's algorithm

```
T := new container for edges
PQ := new min-heap()
start := pick a vertex
PQ.insert(0, start)
for each vertex v != start: PQ.insert(inf, v)
while not PQ.is-empty():
  u := PQ.extract-min()
  T.add((u.pred, u))
  for each v in u's adjacency list:
    if v in PQ and w(u, v) < priority(v):
      PQ.decrease-priority(v, w(u,v))
      v.pred := u
return T
```

Prim's algorithm: time

Let $n = |V|$ and $m = |E|$. Then:

- every vertex enters and leaves min-heap once
  -
  -
- with every edge may call `decrease-priority`
  -
- the rest can be done in $\Theta(1)$ per vertex or per edge

Total time worst case:

Kruskal's algorithm

```
T := new container for edges
L := edges sorted in non-decreasing order by weight
for each vertex v:
  v.cluster := make-cluster(v)
for each (u, v) in L:
  if u.cluster != v.cluster:
    T.add((u,v))
    merge u.cluster and v.cluster
return T
```

Kruskal's algorithm: correctness

Kruskal's algorithm maintains the loop invariants:

1. each cluster is a tree
2. $T \subseteq T_{min}$ for some MST T_{min}

Initially T is empty and clusters are single vertices, so trivially true.

Suppose (1) and (2) are true before line 4.

Kruskal's algorithm: correctness

Suppose (1) and (2) are true before line 4.

Prim's algorithm

```
T := new container for edges
PQ := new min-heap()
start := pick a vertex
PQ.insert(0, start)
for each vertex v != start: PQ.insert(inf, v)
while not PQ.is-empty():
  u := PQ.extract-min()
  T.add((u.pred, u))
  for each v in u's adjacency list:
    if v in PQ and w(u, v) < priority(v):
      PQ.decrease-priority(v, w(u,v))
      v.pred := u
return T
```

Prim's algorithm: correctness

Prim's algorithm maintains the loop invariants:

1. T contains vertices in $V - PQ$
2. for each v in PQ, $priority(v) =$ minimum weight of any edge between v and T
3. $T \subseteq T_{min}$ for some MST T_{min}

Initially T is empty, PQ contains all of V, and all priorities are ∞, so trivially true.

Suppose (1), (2), and (3) are true before line 5.

Prim's algorithm: correctness

Suppose (1), (2), and (3) are true before line 5. Let $p = u.pred$.

General Theorem

Suppose

- $T \subseteq T_{min}$
- can partition V into S and $V - S$ (cut), such that
  - no T edge between V and $V - S$
  - (u, v) is the cheapest edge (light edge) connecting V and $V - S$ (crosses the cut)

Then $T + \{(u, v)\} \subseteq T'_{min}$

- if $(u, v) \notin T_{min}$
- T_{min} has a unique simple path from u to v, via some edge (u', v') with $u' \in S$ and $v' \in V - S$
- T_{min} without (u', v') disconnected; (u, v) would would reconnect
- $weight(u, v) \leq weight(u', v')$
- Choose $T'_{min} = T_{min} - \{(u', v')\} + \{(u, v)\}$