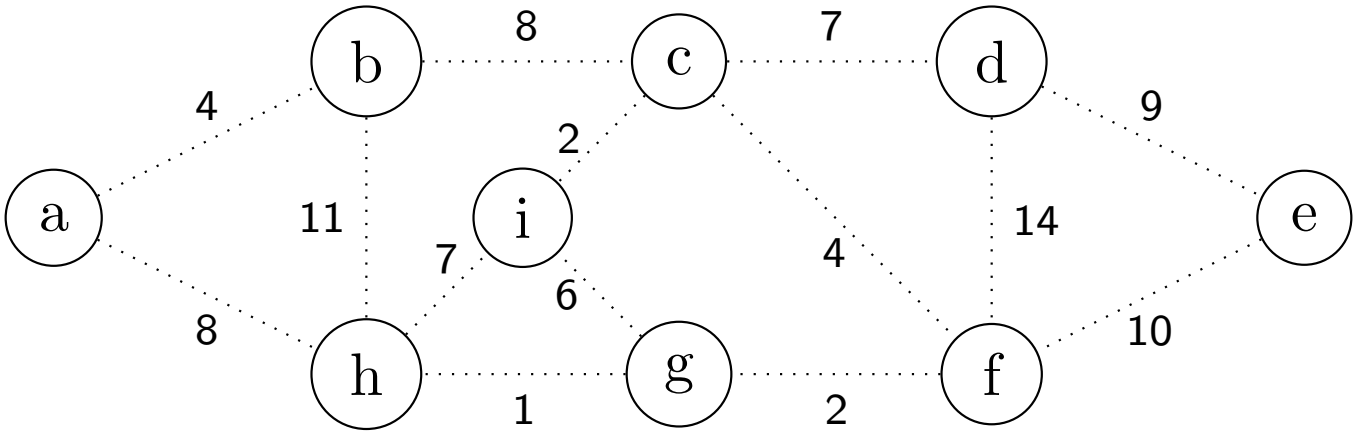
CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1]with huge thanks to Anna Bretscher and Albert Lai

introduction



An (edge-)weighted graph

Applications?

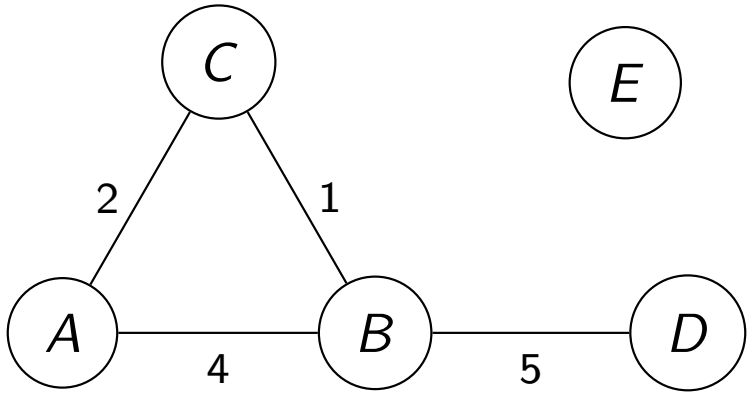
-

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:

edge to itself: 0, no edge: ∞

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

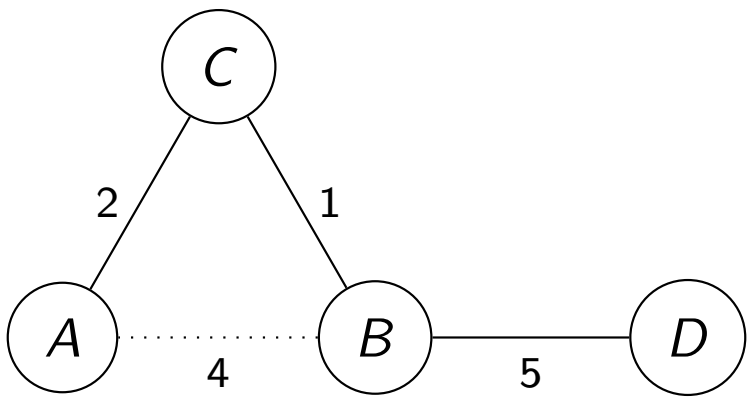
Adjacency lists:

(node, weight)

	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

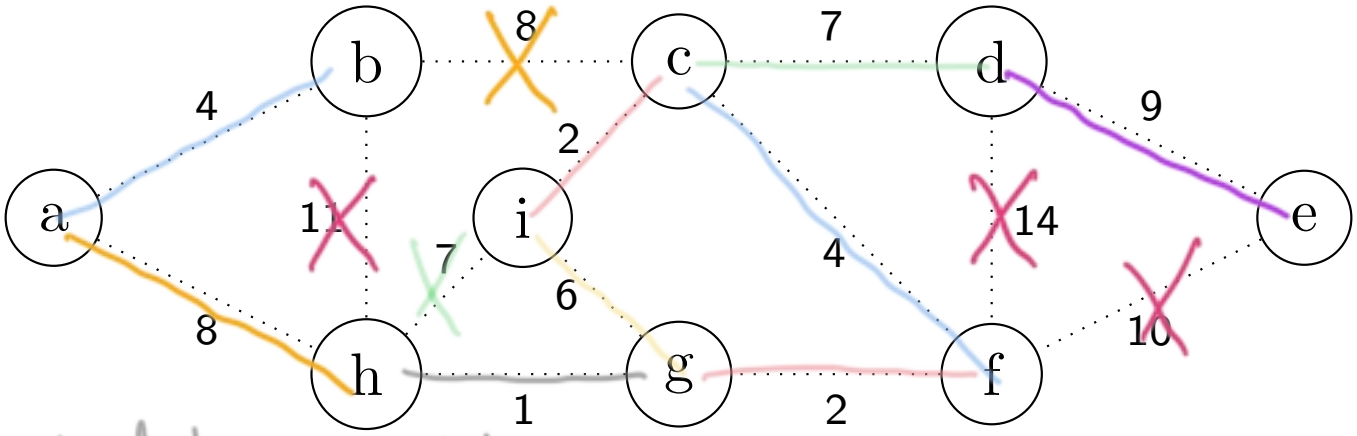
Kruskal's algorithm finds a MST by successive mergers.

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.

merge all elements of cluster

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

Kruskal's algorithm: example



sorted by weights

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: merge(g,h) → merge(c,i) → merge(g,h,f) →

MST: merge(g,h,f,c,i) → merge(a,b) → merge(g,h,f,c,i,d)

→ merge(g,h,f,c,i,d,a,b) → merge(g,h,f,c,i,d,a,b,e)

Kruskal's algorithm

```
0. T := new container for edges
1. L := edges sorted in non-decreasing order by weight
2. for each vertex v:
3.   v.cluster := make-cluster(v)
4. for each (u, v) in L:
5.   if u.cluster != v.cluster:
6.     T.add((u,v))
7.     merge u.cluster and v.cluster
8. 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: smaller list | node
- in the worst case: two list almost same size
- in the worst case: roughly double the size of original list
- then how many such merges can we do?
- each v.cluster is updated at most: $\log_2 |V|$ times

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: $\Theta(m \log m)$ using quicksort/mergesort algorithms
- v.cluster updates: $\Theta(n \log n)$ for n vertices
- the rest is $\Theta(1)$ per vertex or edge

Total: $O(m \log m + n \log n)$

But lets look at n and m:

- maximum number of edges in a graph with n vertices: forest implementation of disjoint sets union by rank with path compression: $O(\log n)$ for each union operation
- then $m \leq \frac{n(n-1)}{2} \leq n^2 \Rightarrow \log m \leq 2 \log n$
  $\Rightarrow \log m \in O(\log n)$

Then total time is $O[(m+n) \log n]$

Faster if faster cluster implementation: later in the course.

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:

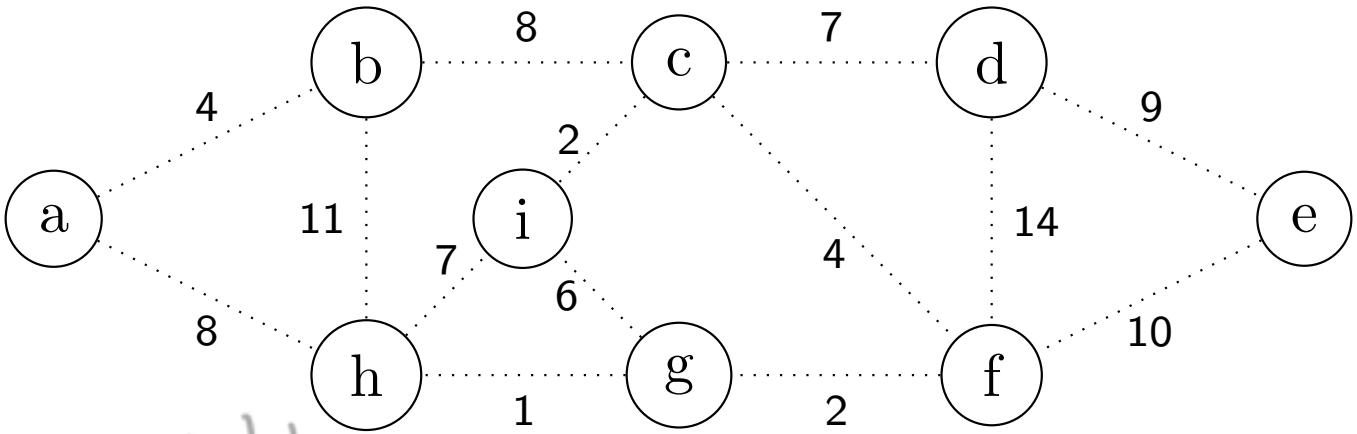
$priority(v) =$ minimum weight of any edge between v and tree

$priority(v) = \infty$ 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 = weight

Priority queue contains vertices *not* in tree:

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

Prim's algorithm

```
0. T := new container for edges
1. PQ := new min-heap()
2. start := pick a vertex
3. PQ.insert(0, start)
4. for each vertex v != start: PQ.insert(inf, v)
5. while not PQ.is-empty():
6.   u := PQ.extract-min()
7.   T.add((u.pred, u))
8.   for each v in u's adjacency list:
9.     if v in PQ and w(u, v) < prioroty(v):
10.      PQ.decrease-priority(v, w(u,v))
11.      v.pred := u
12. return T
```

Prim's algorithm: time

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

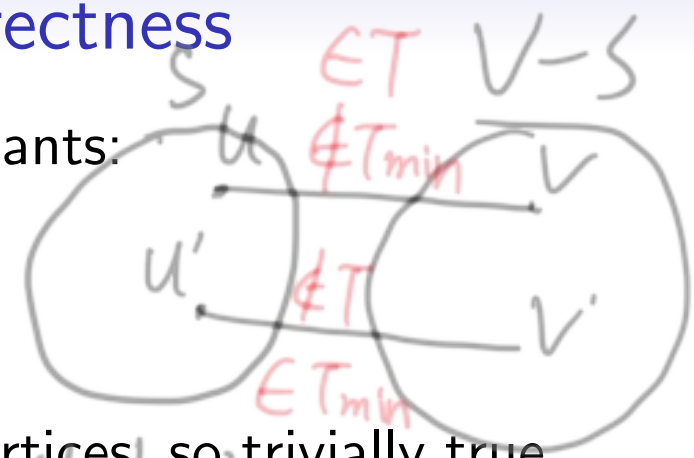
- every vertex enters and leaves min-heap once
  -
  - $O(\log n)$ for each of n vertices
- every edge may trigger a change of priority → decrease-priority
  - $O(\log n)$ for each of m edges
- the rest can be done in $\Theta(1)$ per vertex or per edge

Total time worst case: $O[(m+n)\log n]$

Kruskal's algorithm

```
0. T := new container for edges
1. L := edges sorted in non-decreasing order by weight
2. for each vertex v:
3.   v.cluster := make-cluster(v)
4. for each (u, v) in L:
5.   if u.cluster != v.cluster:
6.     T.add((u,v))
7.     merge u.cluster and v.cluster
8. 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.

$T = \emptyset \subseteq T_{min}$.

u.cluster v.cluster

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

if u.cluster = v.cluster, no changes to (1), (2),

→ (1), (2) holds after line 7

if u.cluster ≠ v.cluster: by assumption, u.cluster and v.clusters trees (distinct vertices, not connected in T)

→ add edge (u, v) and merge clusters, (1) holds

Kruskal's algorithm: correctness

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

if $u.cluster \neq v.cluster$, by assumption, $T \subseteq T_{min}$ for some T_{min} WTS: $T' \subseteq T_{min}$ for some T_{min}

Case 1: $(u,v) \in T_{min}$, $T' = T_{min} \cup \{u,v\} \subseteq T_{min}$

$\rightarrow$ (2) holds after line 7

Case 2: $(u,v) \notin T_{min}$, partition V s.t. $u.cluster \in S$

$v.cluster \in V-S$, no edge connected

T_{min} is MST $\Rightarrow \exists$ unique path from S to $V-S$ in T_{min}

L is sorted let (u,v) before (u',v')

weight $(u,v) \leq$ weight (u',v')

Choose $T'_{min} = T_{min} \setminus \{u',v'\} \cup \{u,v\}$.

disconnects tree by (u',v'), reconnects by (u,v)

$\rightarrow$ (2) holds after line 7

Prim's algorithm

```
0. T := new container for edges
1. PQ := new min-heap()
2. start := pick a vertex
3. PQ.insert(0, start)
4. for each vertex v != start: PQ.insert(inf, v)
5. while not PQ.is-empty():
6.   u := PQ.extract-min()
7.   T.add((u.pred, u))
8.   for each v in u's adjacency list:
9.     if v in PQ and w(u, v) < prioroty(v):
10.      PQ.decrease-priority(v, w(u,v))
11.      v.pred := u
12. 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}

(3) (1) (2)

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

no edges

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

u is dequeued on line 6, (u.pred, u) edge added on line 7, so (1) holds

Prim's algorithm: correctness

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

For arbitrary $v \in PQ$:

① v not adjacent to u, no change to PQ, (2) holds

② v adjacent to u, change priority of v to ensure (2) holds

WTS: adding (u.pred, u) edge to T, $T' \subseteq T_{min}$ for some T_{min}

Case 1: $(u.pred, u) \in T_{min} \Rightarrow T' = T \cup \{u.pred, u\} \subseteq T_{min}$

→ (2) holds after line 11

Case 2: $(u.pred, u) \notin T_{min}$

Just before dequeue u on line 6

must $\exists$ path in T_{min} from $u.pred$ to u

$\Rightarrow$ must $\exists$ $(x, y) \in$ this path s.t. $x \in T, x \notin PQ$, $y \in PQ, y \notin T$

u dequeued before y. $priority(u) \leq priority(y)$

$weight(u.pred, u) \leq weight(x, y)$

then $T'_{min} = T_{min} \setminus \{(x, y)\} \cup \{(u.pred, u)\}$

$\rightarrow$ (2) holds after line 11

General Theorem

Suppose

- $T \subseteq T_{min}$
- can partition V into S and $V - S$ (cut), such that
  - no 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 $T'_{min} = T_{min} - \{(u', v')\} + \{(u, v)\}$