CS 341: Algorithms

University of Waterloo
Éric Schost
eschost@uwaterloo.ca

Module 4: greedy algorithms

Goals

This module: the greedy paradigm through examples

- job scheduling
- interval scheduling
- more scheduling
- fractional knapsack (if time permits)
- Dijsktra's algorithm
- minimum spanning trees

Goals

This module: the greedy paradigm through examples

- job scheduling
- interval scheduling
- more scheduling
- fractional knapsack (if time permits)
- Dijsktra's algorithm
- minimum spanning trees

Computational model:

- word RAM
- assume all weights, capacities, deadlines, etc, fit in a word

Overview

Greedy algorithms

Context: we are trying to solve a **combinatorial optimization** problem:

- have a **large, but finite,** domain $\mathcal{S}$
- want to find an element E in $\mathcal{S}$ that **minimizes / maximizes** a cost function

Greedy algorithms

Context: we are trying to solve a **combinatorial optimization** problem:

- have a **large, but finite,** domain $\mathcal{S}$
- want to find an element E in $\mathcal{S}$ that **minimizes / maximizes** a cost function

Greedy strategy:

- build E step-by-step
- don't think ahead, just try to improve as much as you can at every step
- simple algorithms
- but usually, no guarantee to get the optimal
- it is often hard to prove correctness, and easy to prove incorrectness.

Example: Huffman

Review from CS240: the **Huffman tree**

- we are given **frequencies** $f_1, \ldots, f_n$ **for characters** $c_1, \ldots, c_n$
- we build a **binary tree** for the whole code

Example: Huffman

Review from CS240: the **Huffman tree**

- we are given **frequencies** $f_1, \ldots, f_n$ **for characters** $c_1, \ldots, c_n$
- we build a **binary tree** for the whole code

Greedy strategy: we build the tree **bottom up**.

- create many single-letter trees
- define the **frequency** of a tree as the sum of the frequencies of the letters in it
- build the final tree by putting together smaller trees: **join the two trees with the least frequencies**

Claim: this minimizes $\sum_i f_i \times \{\text{length of encoding of } c_i\}$

A job scheduling problem

The problem

Input:

- n jobs, with processing times $[t(1), \ldots, t(n)]$

The problem

Input:

- n jobs, with processing times $[t(1), \ldots, t(n)]$

Output:

- an ordering of the jobs that minimizes the **sum T of the completions times**
- **completion time:** how long it took **(since the beginning)** to complete a job

The problem

Input:

- n jobs, with processing times $[t(1), \dots, t(n)]$

Output:

- an ordering of the jobs that minimizes the **sum T of the completions times**
- **completion time:** how long it took **(since the beginning)** to complete a job

Example:

- $n = 5$, processing times $[2, 8, 1, 10, 5]$
- in this order,
  $T = 2 \ + \ (8+2) \ + \ (1+8+2) \ + \ (10+1+8+2) \ + \ (5+10+1+8+2) = 70$
- in the order $[1, 2, 5, 8, 10]$,
  $T = 1 \ + \ (2+1) \ + \ (5+2+1) \ + \ (8+5+2+1) \ + \ (10+8+5+2+1) = 54$

Greedy algorithm

Algorithm:

- order the jobs in **non-decreasing** processing times

Greedy algorithm

Algorithm:

- order the jobs in **non-decreasing** processing times

Correctness by an exchange argument

- let $L = [e_1, \ldots, e_n]$ be a permutation of $[1, \ldots, n]$
- suppose that L is **not** in non-decreasing order of processing times. Can it be optimal?
- assumption there exists i such that $\boldsymbol{t(e_i) > t(e_{i+1})}$
- sum of the completion times of L is $\boldsymbol{nt(e_1) + (n-1)t(e_2) + \cdots + t(e_n)}$
- the contribution of e_i and e_{i+1} is $\boldsymbol{(n-i+1)t(e_i) + (n-i)t(e_{i+1})}$
- now, **switch** e_i **and** e_{i+1} **to get a permutation** L'
- their contribution becomes $\boldsymbol{(n-i+1)t(e_{i+1}) + (n-i)t(e_i)}$
- nothing else changes so $\boldsymbol{T(L') - T(L) = t(e_{i+1}) - t(e_i)} < 0$
- so L **not optimal**

Greedy algorithm

Algorithm:

- order the jobs in **non-decreasing** processing times

Review from CS240

- optimal static order for linked list implementation of dictionaries
- same result (up to reverse), same proof

Interval scheduling

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

Output:

- a choice T of intervals that **do not overlap** and that has **maximal cardinality**

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

Output:

- a choice T of intervals that **do not overlap** and that has **maximal cardinality**

Example: A car rental company has the following requests for a given day:

I_1**:** 2pm to 8pm

I_2**:** 3pm to 4pm

I_3**:** 5pm to 6pm

Answer is $T = [I_2, I_3]$.

Template for a greedy algorithm

Greedy($\boldsymbol{I} = [I_1, \ldots, I_n]$)

1. $T \leftarrow [\,]$
2. **while** $\boldsymbol{I}$ is not empty **do**
3. choose an interval I from $\boldsymbol{I}$
4. move I to T
5. remove from $\boldsymbol{I}$ all intervals that overlap with I

Observation: no overlap between the intervals in T

A few attempts

Attempt 1:

- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**

A few attempts

Attempt 1:

- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

A few attempts

Attempt 1:

- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

Attempt 2:

- I is the **shortest interval** in $\boldsymbol{I}$

A few attempts

Attempt 1:

- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

Attempt 2:

- I is the **shortest interval** in $\boldsymbol{I}$
- **no**, for example

A few attempts

Attempt 1:

- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

Attempt 2:

- I is the **shortest interval** in $\boldsymbol{I}$
- **no**, for example

Attempt 3:

- I is the interval in $\boldsymbol{I}$ with the **fewest overlaps**

A few attempts

Attempt 1:

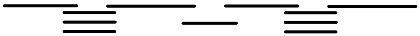
- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

Attempt 2:

- I is the **shortest interval** in $\boldsymbol{I}$
- **no**, for example

Attempt 3:

- I is the interval in $\boldsymbol{I}$ with the **fewest overlaps**
- **no**, for example



A few attempts

Attempt 1:

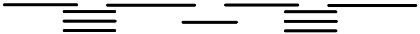
- I is the interval in $\boldsymbol{I}$ with the **earliest starting time**
- **no**, previous example

Attempt 2:

- I is the **shortest interval** in $\boldsymbol{I}$
- **no**, for example

Attempt 3:

- I is the interval in $\boldsymbol{I}$ with the **fewest overlaps**
- **no**, for example



Attempt 4:

- I is the interval in $\boldsymbol{I}$ with the earliest finish time

An $O(n\log(n))$ implementation

Greedy($\boldsymbol{I} = [I_1, \ldots, I_n]$)
1. $T \leftarrow [\,]$
2. sort $\boldsymbol{I}$ by non-decreasing finish time
3. **for** $k = 1, \ldots, n$ **do**
4. if I_k does not overlap the last entry in T
5. append I_k to T

Correctness: greedy stays ahead

Let

- $T = [x_1 < \cdots < x_p]$ be the output of the algorithm,
- $S = [y_1 < \cdots < y_q]$ be any choice of requests without overlaps,
- both sorted by increasing finish time.

Proof that $p \geq q$.

- by induction: for $k = 0, \ldots, q$, $\boldsymbol{p \geq k}$ and $S_k = [x_1 < \cdots < x_k < y_{k+1} < \cdots < y_q]$ has no overlap and is sorted by increasing finish time
- OK for $k = 0$, so we suppose true for some $k < q$, and prove for $k + 1$
- since $[x_1, \ldots, x_k, y_{k+1}]$ is satisfiable, the algorithm didn't stop at x_k. So $\boldsymbol{p \geq k + 1}$.
- by definition of x_{k+1}, **finish**$(x_{k+1}) \leq$ **finish**(y_{k+1}). So we can replace y_{k+1} by x_{k+1} in S_k. We get $S_{k+1} = [x_1 < \cdots < x_{k+1} < y_{k+2} < \cdots < y_q]$, which is still satisfiable and sorted by increasing finish time

Minimizing lateness

The problem

Input:

- jobs $J_1, \ldots, J_n$ with processing times $t(1), \ldots, t(n)$ and deadlines $d(1), \ldots, d(n)$
- can only do one thing at a time

The problem

Input:

- jobs $J_1, \ldots, J_n$ with processing times $t(1), \ldots, t(n)$ and deadlines $d(1), \ldots, d(n)$
- can only do one thing at a time

Output:

- a **scheduling** of the jobs which **minimizes maximal lateness**
  - job J_i starts at time $s(i)$ and finishes at $f(i) = s(i) + t(i)$
  - if $f(i) \geq d(i)$, lateness $\ell(i) = f(i) - d(i)$
- maximal lateness $= \max_i \ell(i)$

Example: 3 jobs

- **prepare my slides:** need $t(1) = 4$ hours, deadline $d(1) = 2$ hours
- **write solutions to assignments:** need $t(2) = 6$ hours, deadline $d(2) = 1$ hour
- **finish the midterm:** need $t(3) = 10$ hours, deadline $d(3) = 24$ hours

Example: 3 jobs

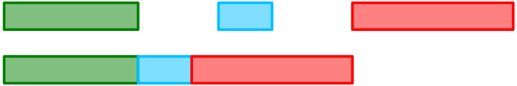
- **prepare my slides:** need $t(1) = 4$ hours, deadline $d(1) = 2$ hours
- **write solutions to assignments:** need $t(2) = 6$ hours, deadline $d(2) = 1$ hour
- **finish the midterm:** need $t(3) = 10$ hours, deadline $d(3) = 24$ hours

- **1, then 2, then 3:** latenesses $[2, 9, 0]$
- **2, then 1, then 3:** latenesses $[8, 5, 0]$ (optimal)

No breaks

Observation:

- if a scheduling has **idle time**, we can improve it by removing the breaks



- so the optimal has no idle time, and is given by an **ordering** of the jobs

A few attempts

Attempt 1:

- do short jobs first

A few attempts

Attempt 1:

- do short jobs first
- **no**, last example

A few attempts

Attempt 1:

- do short jobs first
- **no**, last example

Attempt 2:

- do jobs with little slack first   slack $= d(i) - t(i)$

A few attempts

Attempt 1:

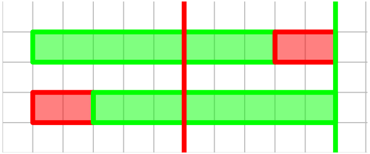
- do short jobs first
- **no**, last example

Attempt 2:

- do jobs with little slack first

slack $= d(i) - t(i)$

- **no**



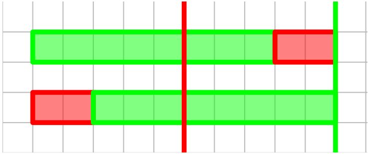
A few attempts

Attempt 1:

- do short jobs first
- **no**, last example

Attempt 2:

- do jobs with little slack first $\quad$ slack $= d(i) - t(i)$
- **no**



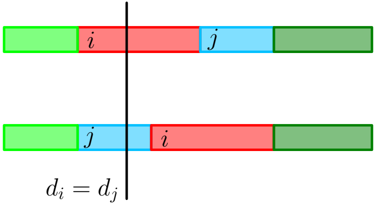
Attempt 3:

- do jobs in non-decreasing deadline order

Non-uniqueness

Observation:

- if $d(i) = d(j)$, the orderings $[\ldots, \boldsymbol{i}, \boldsymbol{j}, \ldots]$ and $[\ldots, \boldsymbol{j}, \boldsymbol{i}, \ldots]$ have the same max-lateness (because the second job is the latest)
- so **all** orderings in non-decreasing deadline order have the same max-lateness



Non-uniqueness

Observation:

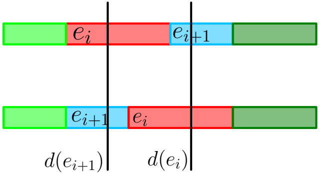
- if $d(i) = d(j)$, the orderings $[\dots, \boldsymbol{i}, \boldsymbol{j}, \dots]$ and $[\dots, \boldsymbol{j}, \boldsymbol{i}, \dots]$ have the same max-lateness (because the second job is the latest)
- so **all** orderings in non-decreasing deadline order have the same max-lateness

Definition:

- an inversion in $L = [e_1, \dots, e_n]$ is a pair (i, j) with $i < j$ and $d(e_i) > d(e_j)$
- L has no inversion $\iff$ L in non-decreasing deadline order

Correctness: exchange argument

- let $L = [e_1, \ldots, e_n]$ be a solution (as a permutation of $[1, \ldots, n]$)
- suppose that L is **not** in non-decreasing order of deadlines, so there exists i such that $\boldsymbol{d(e_i) > d(e_{i+1})}$
- now, **switch** e_i **and** e_{i+1} **to get a permutation** L'
- the lateness of e_{i+1} cannot increase (because we do e_{i+1} earlier than before), so at most max_lateness(L)
- the **new** lateness of e_i is **at most** the **old** lateness of e_{i+1}, so at most max_lateness(L)



Correctness: exchange argument

- let $L = [e_1, \ldots, e_n]$ be a solution (as a permutation of $[1, \ldots, n]$)
- suppose that L is **not** in non-decreasing order of deadlines, so there exists i such that $\boldsymbol{d(e_i) > d(e_{i+1})}$
- now, **switch** e_i **and** e_{i+1} **to get a permutation** L'
- the lateness of e_{i+1} cannot increase (because we do e_{i+1} earlier than before), so at most max_lateness(L)
- the **new** lateness of e_i is **at most** the **old** lateness of e_{i+1}, so at most max_lateness(L)
- nothing else changes, so max_lateness$(L') \leq$ max_lateness(L)
- and we have removed an inversion
- keep going: after at most $n(n-1)/2$ iterations, we have L_{ord} with **no inversion** and such that **max_lateness$(L_{\text{ord}}) \leq$ max_lateness(L)**

Interval coloring

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \ldots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

The problem

Input:

- n intervals $I_1 = [s_1, f_1], \dots, I_n = [s_n, f_n]$ start time, finish time
- also write $s_j = \textbf{start}(I_j)$, $f_j = \textbf{finish}(I_j)$

Output:

- assignment of **colors** to each interval
- overlapping intervals get **different colors**
- **minimize** the number of colors used overall

Remarks:

- another version: finding classrooms for lectures
- colors $\leftrightarrow$ numbers $1, 2, \dots$
- $\textbf{finish}(I_j) = \textbf{start}(I_k)$ not an overlap

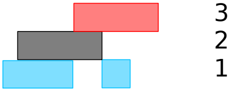
A blueprint for a greedy algorithm

GreedyColoring($\boldsymbol{I} = [I_1, \ldots, I_n]$)
1. sort $\boldsymbol{I}$ somehow
2. **for** $k = 1, \ldots, n$ **do**
3. color I_k with the **minimum** color not used by any of the previous intervals that overlap I_k

A few attempts

Attempt 1:

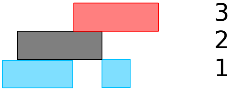
- sort by non-decreasing finish times
- no



A few attempts

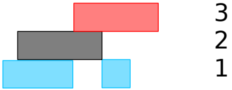
Attempt 1:

- sort by non-decreasing finish times
- no



Attempt 2:

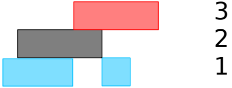
- sort from shortest to longest
- no



A few attempts

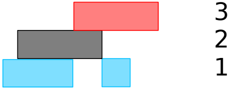
Attempt 1:

- sort by non-decreasing finish times
- no



Attempt 2:

- sort from shortest to longest
- no



Attempt 3:

- sort by non-decreasing starting times
- maybe

Correctness

Claim

- we suppose the algorithm uses $\boldsymbol{k}$ colors
- we prove that we can't use fewer.

Correctness

Claim

- we suppose the algorithm uses $\boldsymbol{k}$ colors
- we prove that we can't use fewer.

Proof

- suppose we color I_t with color k
- so I_k overlaps with $k-1$ intervals, say $I_{\alpha_1}, \ldots, I_{\alpha_{k-1}}$ seen previously
- so for all j, $\boldsymbol{s_{\alpha_j} \leq s_t < f_{\alpha_j}}$
- so there is a little interval $[s_t, s_t + \varepsilon]$ in all I_{α_j} and I_t
- so we can't do with less than k colors

Exercise

Give an $\boldsymbol{O(n \log(n))}$ implementation.

Fractional knapsack

The problem

Input:

- items $I_1, \ldots, I_n$ with weights $\boldsymbol{w_1, \ldots, w_n}$ and positive values $\boldsymbol{v_1, \ldots, v_n}$
- a capacity W

Output:

- fractions $E = e_1, \ldots, e_n$ such that
  - $0 \leq e_j \leq 1$ for all j
  - $e_1 w_1 + \cdots + e_n w_n \leq W$
  - $e_1 v_1 + \cdots + e_n v_n$ maximal

Example:

- $\boldsymbol{w_1 = 10, v_1 = 60}$, $\boldsymbol{w_2 = 30, v_2 = 90}$, $\boldsymbol{w_3 = 20, v_3 = 100}$
- $\boldsymbol{W = 50}$
- optimal is $\boldsymbol{e_1 = 1}$, $\boldsymbol{e_2 = 2/3}$, $\boldsymbol{e_3 = 1}$, total value **220**

The problem

Input:

- items $I_1, \dots, I_n$ with weights $\boldsymbol{w_1, \dots, w_n}$ and positive values $\boldsymbol{v_1, \dots, v_n}$
- a capacity W

Output:

- fractions $E = e_1, \dots, e_n$ such that
  - $0 \leq e_j \leq 1$ for all j
  - $e_1 w_1 + \cdots + e_n w_n \leq W$
  - $e_1 v_1 + \cdots + e_n v_n$ maximal

Remark:

- **0/1-version:** $e_j \in \{0, 1\}$ for all j
- dynamic programming

The knapsack should be full

Remark:

- if $\sum_i w_i < W$, just take all $e_i = 1$
- so assume $\boldsymbol{\sum_i w_i \geq W}$

The knapsack should be full

Remark:

- if $\sum_i w_i < W$, just take all $e_i = 1$
- so assume $\boldsymbol{\sum_i w_i \geq W}$

Observation:

- suppose we have an assignment with $\boldsymbol{\sum_i e_i w_i < W}$
- then some e_i must be **less than** 1
- so we can increase the value by non-decreasing this e_i

The knapsack should be full

Remark:

- if $\sum_i w_i < W$, just take all $e_i = 1$
- so assume $\sum_i w_i \geq W$

Observation:

- suppose we have an assignment with $\sum_i e_i w_i < W$
- then some e_i must be **less than** 1
- so we can increase the value by non-decreasing this e_i

Consequence:

- it is enough to consider assignments with $\sum_i e_i w_i = W$

A few attempts

Attempt 1:

- pack with items in **decreasing value** v_i

A few attempts

Attempt 1:

- pack with items in **decreasing value** v_i
- **no**, previous example (we get $[0, 1, 1]$ with total value **190**)

A few attempts

Attempt 1:

- pack with items in **decreasing value** v_i
- **no**, previous example (we get $[0, 1, 1]$ with total value $\mathbf{190}$)

Attempt 2:

- pack with items in **increasing weight** w_i

A few attempts

Attempt 1:

- pack with items in **decreasing value** v_i
- **no**, previous example (we get $[0, 1, 1]$ with total value **190**)

Attempt 2:

- pack with items in **increasing weight** w_i
- **no**: $\boldsymbol{W = 10}$, $\boldsymbol{w_1 = 10, v_1 = 1}$, $\boldsymbol{w_2 = 5, v_2 = 100}$

A few attempts

Attempt 1:

- pack with items in **decreasing value** v_i
- **no**, previous example (we get $[0, 1, 1]$ with total value $\mathbf{190}$)

Attempt 2:

- pack with items in **increasing weight** w_i
- **no**: $\boldsymbol{W = 10}$, $\boldsymbol{w_1 = 10, v_1 = 1}$, $\boldsymbol{w_2 = 5, v_2 = 100}$

Attempt 3:

- pack with items in **decreasing "value per kilo"** v_i/w_i
- first example $[\mathbf{6, 3, 5}]$, second example $[\mathbf{1/10, 20}]$

Pseudo-code

```
GreedyKnapsack(v, w, W)
    E ← [0, ..., 0]
    sort items by decreasing order of v_i/w_i
    for k = 1, ..., n do
        if w_k < W then
            E[k] ← 1
            W ← W − w_k
        else
            E[k] ← W/w_k
            return
```

Remark: output is $S = [1, \ldots, 1, e_k, 0, \ldots, 0]$

Runtime: $O(n \log(n))$

Correctness: exchange argument

- let $E = [e_1, \ldots, e_n]$ be the optimal assignment, with $\boldsymbol{\sum e_i w_i = W}$
- let $S = [s_1, \ldots, s_n]$ be **any** assignment, with $\boldsymbol{\sum s_i w_i = W}$
- suppose S different from E, and let i be the **first** index with $\boldsymbol{e_i \neq s_i}$
- greedy strategy: $\boldsymbol{e_i > s_i}$
- because their weights are the same, there is $j > i$ with $\boldsymbol{s_j > e_j}$
- set $\boldsymbol{s'_i = s_i + \alpha/w_i}$ and $\boldsymbol{s'_j = s_j - \alpha/w_j}$, for α TBD > 0, all other $s'_k = s_k$
- in any case, $\boldsymbol{\sum s'_i w_i = W}$ and **value$(S') \geq$ value(S)**
- choose α such that **either** $s'_i = e_i$ **or** $s'_j = e_j$

$$\alpha = \min(w_i(e_i - s_i), w_j(s_j - e_j))$$

- so we found S' that has **one more common entry** with E, and which is at least as good as S
- keep going

Dijkstra's algorithm

Conventions

Input:

- a **directed** graph $G = (V, E)$
- with **weights** $w(e)$ on the edges
  $w(\gamma)$ = weight of a path γ = sum of the weights of its edges
- no **loops** = edges $v \to v$
- no **isolated vertices**, with no incoming or outgoing edge $m \geq n/2$

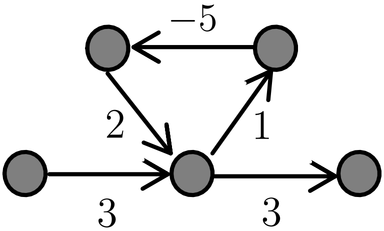
Output:

- the shortest (=minimal weight) paths between a **source** s and **all vertices**
- dynamic programming: shortest paths between **all vertices**

Remark: nothing faster known (to me) for single-source, single-destination

Remarks

1. shortest paths may not exist if there are **negative length cycles**

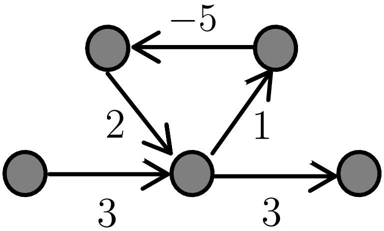


some algorithms can deal with negative edges (and detect negative cycles)

Dijkstra's algorithm needs positive weights

Remarks

1. shortest paths may not exist if there are **negative length cycles**



some algorithms can deal with negative edges (and detect negative cycles)

Dijkstra's algorithm needs positive weights

2. if there exists a shortest path $s \rightsquigarrow t$, write $\boldsymbol{\delta(s,t)}$ for its weight
   - called the **distance** from s to t (but we may not have $\delta(s,t) = \delta(t,s)$)
   - if there is **no path** $s \rightsquigarrow t$, $\delta(s,t) = \infty$

Outlook

Assumption

All weights are non-negative

Outlook

Assumption

All weights are non-negative

Idea of the algorithm:

- starting from s, grow a tree (S, T) rooted at s, together with the **distances** $\delta(s, v)$ for v in S
- at every step, add to S the remaining vertex v **closest to s**
- **no negative weight:** this vertex is on an edge (u, v), u in S, v in $V - S$
- if there is no such edge, we're done (all remaining vertices are unreachable)

greedy algorithm!

Key property

Claim

Let (S, T) be a tree rooted at s and take an edge (u, v) such that

- u is in S, v is in $V - S$
- $\delta(s, u) + w(u, v)$ **minimal** among these edges

Then $\boldsymbol{\delta(s, u) + w(u, v) = \delta(s, v)}$

Key property

Claim

Let (S, T) be a tree rooted at s and take an edge (u, v) such that

- u is in S, v is in $V - S$
- $\delta(s, u) + w(u, v)$ **minimal** among these edges

Then $\boldsymbol{\delta(s, u) + w(u, v) = \delta(s, v)}$

Proof:

- take a path $\gamma : s \rightsquigarrow v$ and let (x, y) be its first edge $\boldsymbol{S \to V - S}$
- $w(\gamma) = w(\boldsymbol{s \rightsquigarrow x}) + w(x, y) + w(\boldsymbol{y \rightsquigarrow v}) \geq \delta(s, x) + w(x, y) \boldsymbol{+0}$
- so $\boldsymbol{w(\gamma) \geq \delta(s, u) + w(u, v)}$ — choice of u, v
- but also $\boldsymbol{\delta(s, u) + w(u, v) \geq \delta(s, v)}$ — def of distance $s \to v$
- take **shortest** $\boldsymbol{\gamma}$: $w(\gamma) = \delta(s, v)$ so $\boldsymbol{\delta(s, v) \geq \delta(s, u) + w(u, v) \geq \delta(s, v)}$

High-level view of the algorithm

Dijkstra(G, s)

$S \leftarrow \{s\}$
while $S \neq V$ **do**
    choose (u, v) with u in S, v not in S and $\delta(s, u) + w(u, v)$ minimal (the min value gives $\delta(s, v)$)
    add v to S
    if not such (u, v), **stop**

High-level view of the algorithm

```
Dijkstra(G, s)
    S ← {s}
    while S ≠ V do
        choose (u, v) with u in S, v not in S and δ(s, u) + w(u, v) minimal
        (the min value gives δ(s, v))
        add v to S
        if not such (u, v), stop
```

Correctness:

- we find $\delta(s, v)$ for all v in S
- if $S = V$ at the end, OK
- if not, when we stop, the remaining vertices are unreachable

High-level view of the algorithm

Dijkstra(G, s)
1. $S \leftarrow \{s\}$
2. **while** $S \neq V$ **do**
3. choose (u, v) with u in S, v not in S and $\delta(s, u) + w(u, v)$ minimal (the min value gives $\delta(s, v)$)
4. add v to S
5. **if** not such (u, v), **stop**

Correctness:

- we find $\delta(s, v)$ for all v in S
- if $S = V$ at the end, OK
- if not, when we stop, the remaining vertices are unreachable

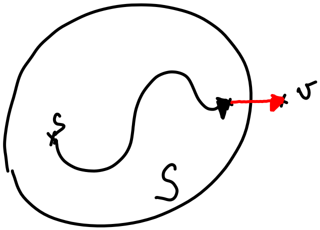
Questions:

- how to find (u, v) efficiently
- probably need a priority queue (heap) of some kind
- good choice: a priority queue of vertices

The min-priority queue

Building P

- contains all vertices in $\boldsymbol{V - S}$ (initially, all V)
- set $\mathsf{priority}[s] = 0$
- for $v \neq s$, we will maintain $\mathsf{priority}[v] = \boldsymbol{\min_{u \in S, (u,v) \in E}(\delta(s,u) + w(u,v))}$ with $\min(\emptyset) = \infty$



- initially $\mathsf{priority}[v] = \infty$ for $v \neq s$
- also store the vertex u that gives the min

The min-priority queue

Updating P

- if v is the vertex with **minimal priority**, then

$$\begin{aligned}\mathsf{priority}[v] &= \min_{v' \in V - S} \mathsf{priority}[v'] \\ &= \min_{v' \in V - S} \min_{u \in S, (u,v') \in E} (\delta(s,u) + w(u,v')) \\ &= \delta(s,v) \qquad \text{(key property)}\end{aligned}$$

we store it in an array d

The min-priority queue

Updating P

- if v is the vertex with **minimal priority**, then

$$\begin{aligned}\mathsf{priority}[v] &= \min_{\boldsymbol{v' \in V - S}} \mathsf{priority}[v'] \\ &= \min_{\boldsymbol{v' \in V - S}} \min_{\boldsymbol{u \in S, (u, v') \in E}} (\delta(s, u) + w(u, v')) \\ &= \delta(s, v) \qquad \text{(key property)}\end{aligned}$$

  we store it in an array d

- then for all v' remaining in P, we must set

$$\mathsf{priority}[v'] = \boldsymbol{\min_{u \in S + v, (u, v') \in E} (\delta(s, u) + w(u, v'))}$$

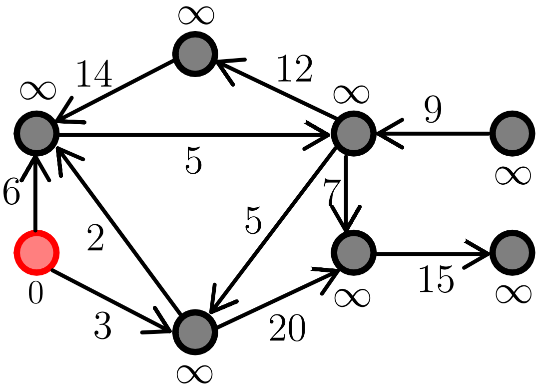
  - if there is no edge (v, v'), $\mathsf{priority}[v']$ unchanged
  - else, the new priority is $\min(\mathsf{priority}[v'], d[v] + w(v, v'))$

Pseudo-code

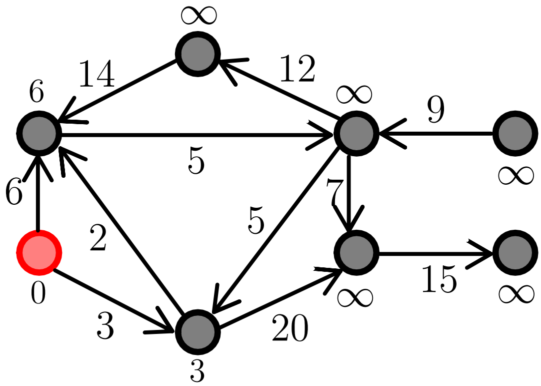
Dijkstra(G, s)

$P \leftarrow$ **heapify**$([s, 0, s], [v, \infty, \bullet]_{v \neq s})$
while P not empty **do**
    $[v, \ell, u] \leftarrow$ **remove_min**(P)
    $d[v] \leftarrow \ell$
    parent$[v] \leftarrow u$
    for all edges (v, v') **do**
        if $d[v] + w(v, v') <$ priority$[v']$ **then**
            replace $[v', \_, \_]$ by $[v', d[v] + w(v, v'), v]$ in P

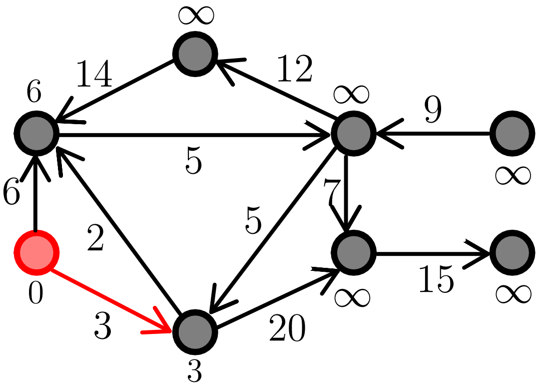
Example



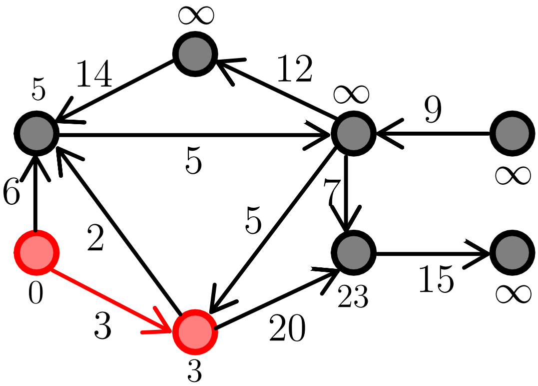
Example



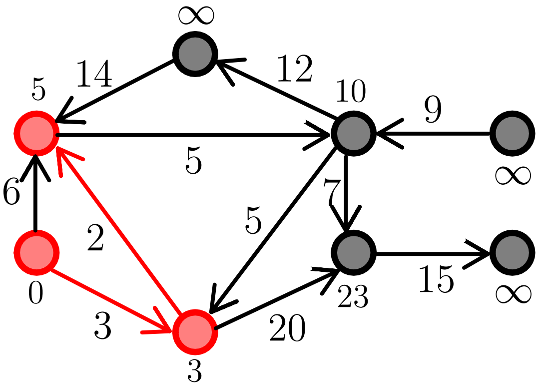
Example



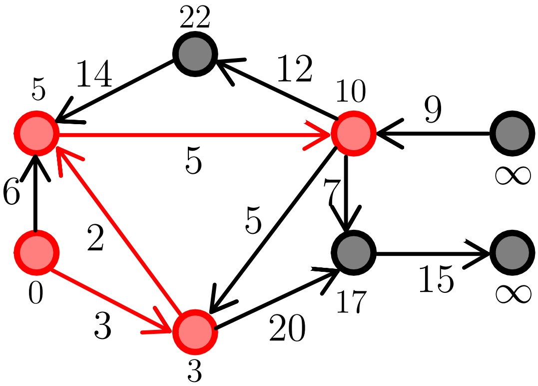
Example



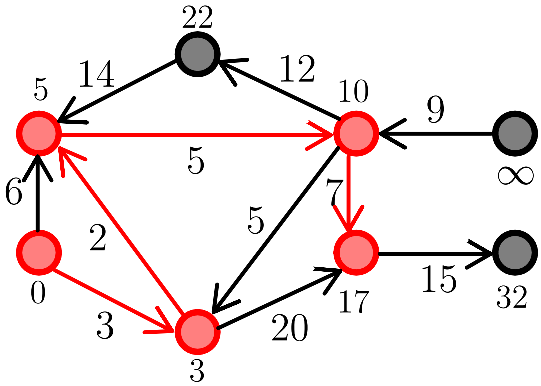
Example



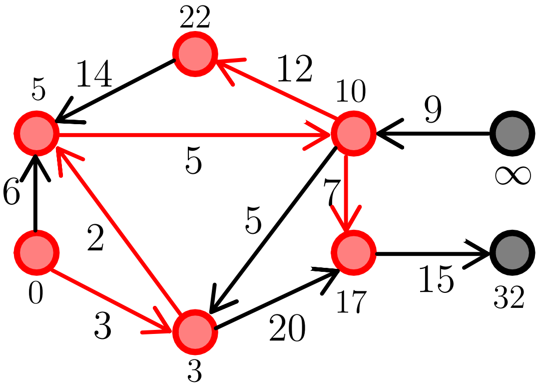
Example



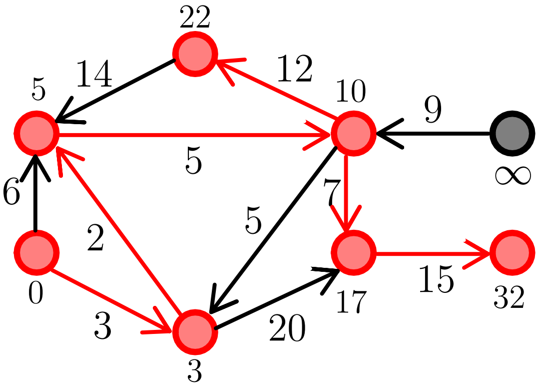
Example



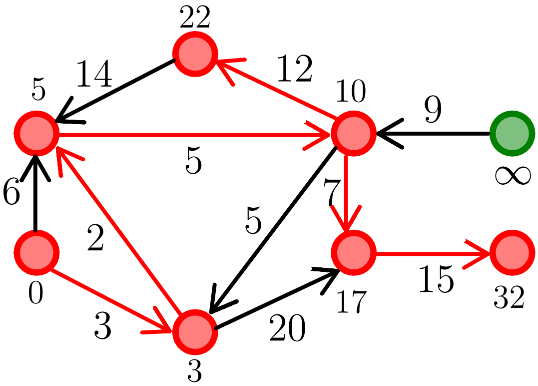
Example



Example



Example



Runtime

Enhanced priority queue

- we need to be able to **change the priority** of a key
- binary heap implementation: $\boldsymbol{O(\log(n))}$ for remove-min and change priority

Total

- n remove min, m change priority $\quad m \geq n/2$
- gives $\boldsymbol{O(m\log(m))}$ $\quad \log(m) \in \Theta(\log(n))$

Remark

- **Fibonacci heaps:** constant amortized time for change priority
- total becomes $\boldsymbol{O(m + n\log(m))}$

Kruskal's algorithm

Spanning trees

Definition:

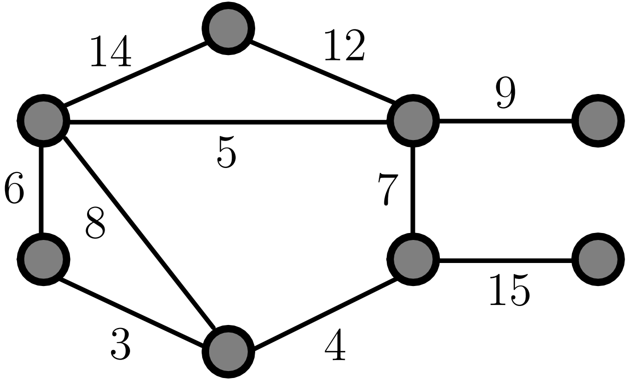
- $G = (V, E)$ is a **connected graph**
- a **spanning tree** in G is a tree of the form (V, T), with T a subset of E
- **in other words:** a tree with edges from E that covers all vertices
- examples: BFS tree, DFS tree

Now, suppose the edges have **weights** $w(e_i)$

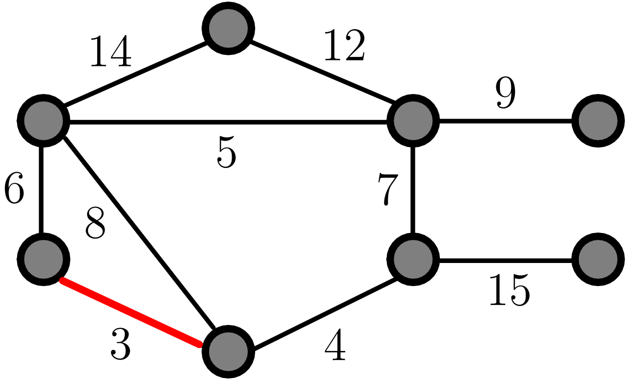
Goal:

- a spanning tree with **minimal weight**

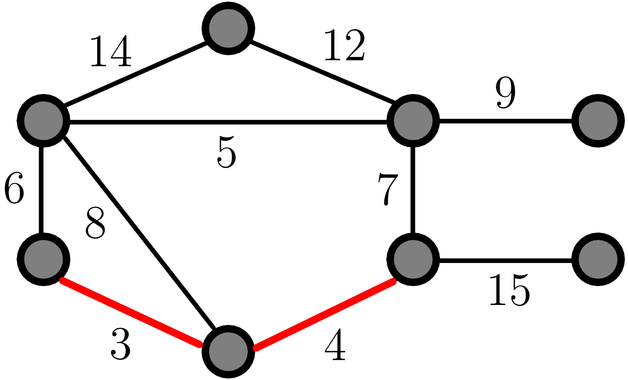
Example



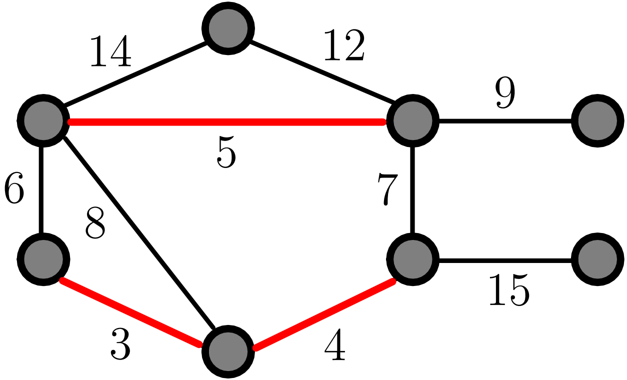
Example



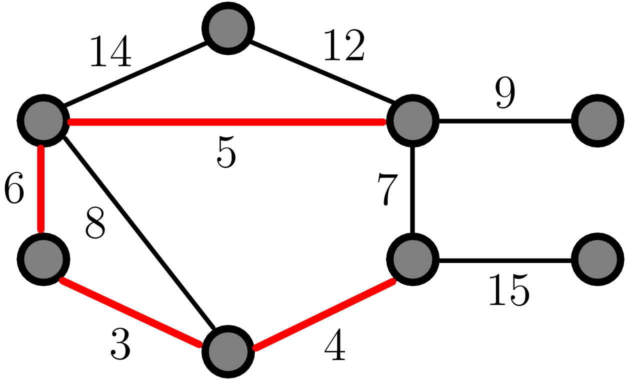
Example



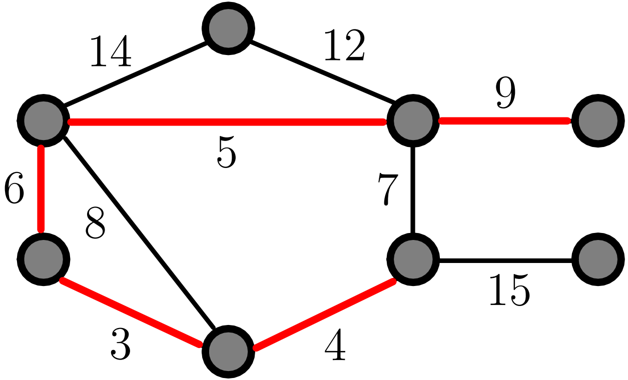
Example



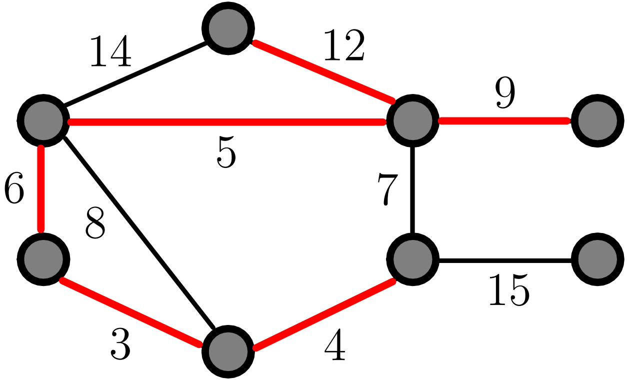
Example



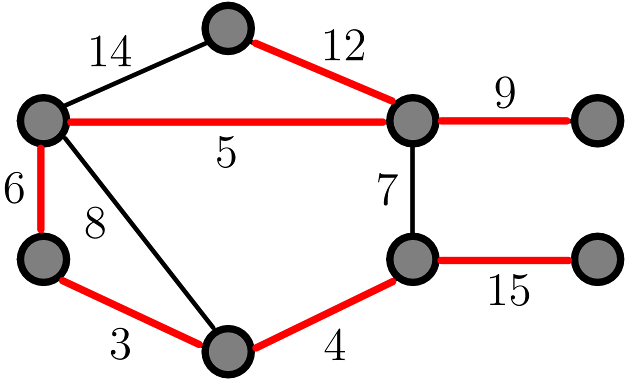
Example



Example



Example



Kruskal's algorithm

GreedyMST(G)

$A \leftarrow [\ ]$
sort edges by non-decreasing weight
for $k = 1, \ldots, m$ **do**
    if e_k does not create a cycle in A **then**
        append e_k to A

Properties of the output

Claim

If the output is $A = [e_1, \ldots, e_r]$, then $\boldsymbol{(V, A)}$ **is a spanning tree** (and so $r = n - 1$)

Proof:

- of course, (V, A) has no cycle: it is a **union of trees**
- suppose (V, A) is **not connected**. Then, there exists an edge e not in A, such that $(V, A \cup \{e\})$ still has no cycle (joining two connected components)
- when we checked e, we did not include it
- means that it created a loop with some edges already in A: **impossible**.

Adding edges to spanning trees

Claim

Let (V, A) be a spanning tree, and let e be an edge not in A.

Then adding e to A creates **a unique cycle**

Proof (bonus)

- let $e = \{v, w\}$.
- **from 239:** in (V, A), there is a **unique simple path** $\boldsymbol{\gamma : v \rightsquigarrow w}$
- adding e creates a cycle
- if it created two different cycles, there would be two paths in (V, A)

Exchanging edges

Claim

Let (V, A) and (V, T) be two spanning trees, and let e be an edge in T but not in A.

- there exists an edge e' in A but not in T such that $(V, T + e' - e)$ is still a spanning tree
- e' is on the cycle that e creates in A.

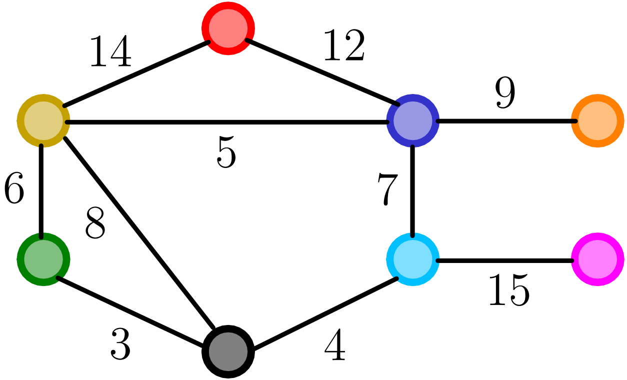
Proof (bonus):

- write $e = \{v, w\}$
- $(V, A + e)$ contains a cycle $\boldsymbol{c = v, w, \ldots, v}$
- removing e from T splits $(V, T - e)$ into two connected components T_1, T_2
- $\boldsymbol{c}$ starts in T_1, crosses over to T_2, so it contains another edge e' between T_2 and T_1
- e' is in A, but not in T
- $(V, T + e' - e)$ is a spanning tree (covers V, $n - 1$ edges, connected)

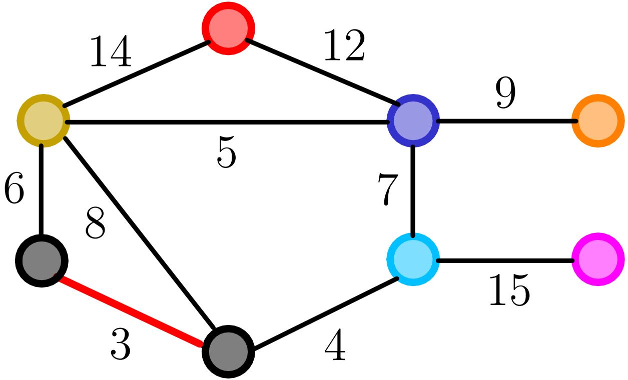
Correctness: exchange argument

- let A be the output of the algorithm
- let (V, T) be **any** spanning tree
- if $T \neq A$, let e be an edge in T but not in A
- so there is an edge e' in A but not in T such that $\boldsymbol{(V, T + e' - e)}$ is a spanning tree, **and** e' is on the cycle that e creates in A
- during the algorithm, we considered e but rejected it, because it created a cycle in A
- all other elements in this cycle have smaller (or equal) weight
- so $\boldsymbol{w(e') \leq w(e)}$
- so $T' = T + e' - e$ has weight $\leq w(T)$, and **one more common element** with A
- keep going

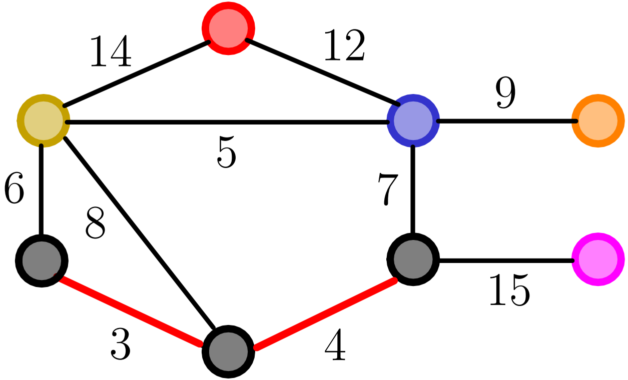
Merging connected sets of vertices



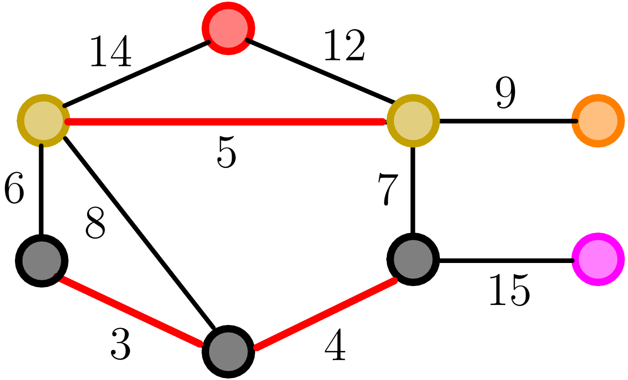
Merging connected sets of vertices



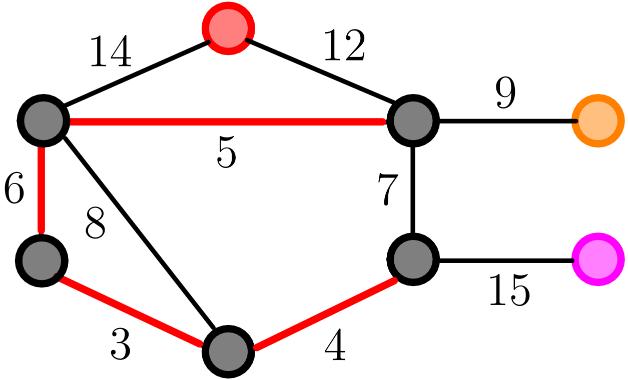
Merging connected sets of vertices



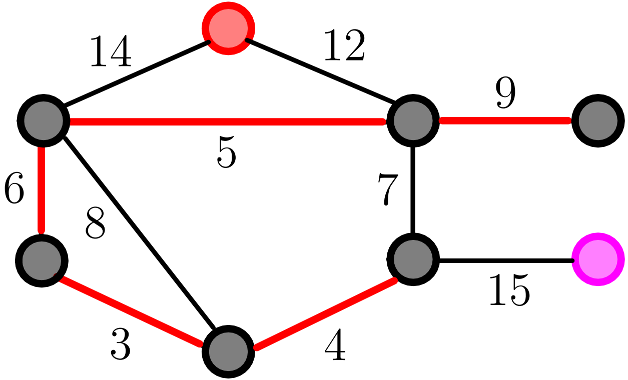
Merging connected sets of vertices



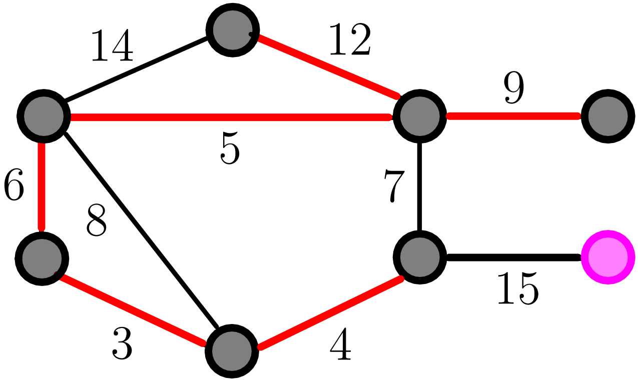
Merging connected sets of vertices



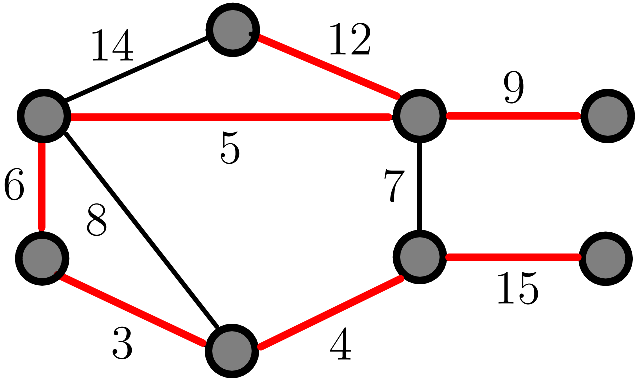
Merging connected sets of vertices



Merging connected sets of vertices



Merging connected sets of vertices



Data structures

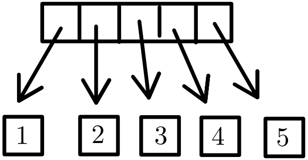
Operations on **disjoint sets of vertices:**

- **Find:** identify which set contains a given vertex
- **Union:** replace two sets by their union

GreedyMST_UnionFind(G)
$T \leftarrow [\ ]$
$U \leftarrow \{\{v_1\}, \ldots, \{v_n\}\}$
sort edges by non-decreasing weight
for $k = 1, \ldots, m$ **do**
  if U.Find(e_k.1) $\neq$ U.Find(e_k.2) **then**
    U.Union(U.Find(e_k.1), U.Find(e_k.2))
    append e_k to T

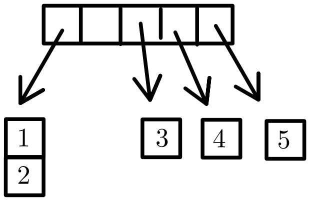
An OK solution

- U is an **array of linked lists**



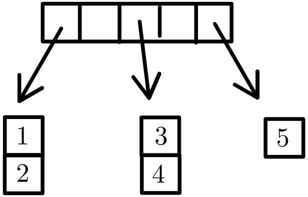
An OK solution

- U is an **array of linked lists**



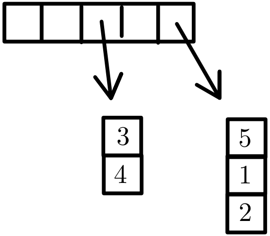
An OK solution

- U is an **array of linked lists**



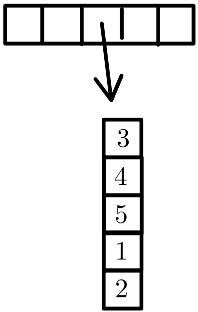
An OK solution

- U is an **array of linked lists**



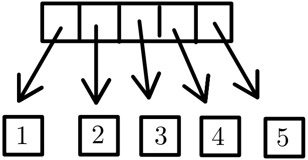
An OK solution

- U is an **array of linked lists**



An OK solution

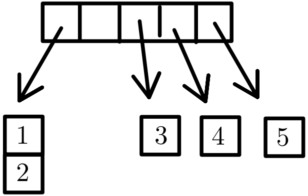
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [1, 2, 3, 4, 5]$$

An OK solution

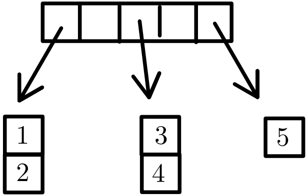
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [1, 1, 3, 4, 5]$$

An OK solution

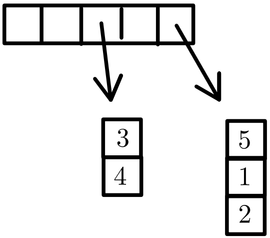
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [1, 1, 3, 3, 5]$$

An OK solution

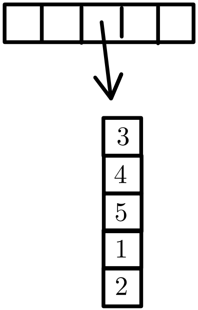
- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [5, 5, 3, 3, 5]$$

An OK solution

- U is an **array of linked lists**
- to do find, add an **array of indices**, $X[i]$ = set that contains i



$$X = [3, 3, 3, 3, 3]$$

Analysis

Worst case:

- **Find** is $O(1)$
- **Union** **traverses** one of the linked lists, **updates** corresponding entries of X, concatenates two linked lists. Worst case $\Theta(n)$

Analysis

Worst case:

- **Find** is $O(1)$
- **Union traverses** one of the linked lists, **updates** corresponding entries of X, concatenates two linked lists. Worst case $\Theta(n)$

Kruskal's algorithm:

- sorting edges $\boldsymbol{O(m\log(m))}$
- $O(m)$ **Find**
- $O(n)$ **Union**

Worst case $\boldsymbol{O(m\log(m)+n^2)}$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **tail** of the lists to concatenate in $O(1)$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **tail** of the lists to concatenate in $O(1)$

Key observation: worst case for **one** union **still** $\Theta(n)$, but better total time.

- for any given vertex v, the size of the set containing V **at least doubles** when we update $X[v]$
- so $X[v]$ updated at most $\log(n)$ times
- so the **total** cost of union per vertex is $O(\log(n))$

A simple heuristics for Union

Modified Union

- each set in U keeps track of its size
- only traverse the **smaller list**
- also add a pointer to the **tail** of the lists to concatenate in $O(1)$

Key observation: worst case for **one** union **still** $\Theta(n)$, but better total time.

- for any given vertex v, the size of the set containing V **at least doubles** when we update $X[v]$
- so $X[v]$ updated at most $\log(n)$ times
- so the **total** cost of union per vertex is $O(\log(n))$

Conclusion: $\boldsymbol{O(n \log(n))}$ for all unions and $\boldsymbol{O(m \log(m))}$ total