11 Greedy Algorithms

14 January 2025

Prof. Dr. Sebastian Wild

CS566 (Wintersemester 2024/25)
Philipps-Universität Marburg

version `2025-01-20 16:03`

Learning Outcomes

Unit 11: ***Greedy Algorithms***

1. Describe informally what greedy algorithms are.
2. Know exemplary problems and their greedy solutions: Change-Making Problem, MSTs, SSSPP, Assignment Problem.
3. Be able to design and proof correctness of greedy algorithms for (simple) algorithmic problems.
4. Be able to explain the matroid properties and its relation to greedy algorithms.

Outline

11 Greedy Algorithms

11.1 Introduction
11.2 How Can Greed Succeed?
11.3 Greed in Graphs I: MSTs
11.4 Greed in Graphs II: Prim's MST Algorithm
11.5 Greed in Graphs III: Shortest Paths
11.6 Greedy Schedules
11.7 The Essence of Greed: Matroids

11.1 Introduction

Myopic Optimization

- In a *"greedy" algorithm*,
  we assemble a solution to an **optimization** problem **step by step**
  always picking the next step to maximize **current** gain,
  and we **never take back** earlier steps.

"Take what you can, give nothing back!"

Myopic Optimization

- In a *"greedy" algorithm*,
  we assemble a solution to an **optimization** problem **step by step**
  always picking the next step to maximize **current** gain,
  and we **never take back** earlier steps.

"Take what you can, give nothing back!"

- reminiscent of *gradient-descent* algorithms
  but discrete and even more unwilling to undo mistakes

⇝ greedy algorithms only yield optimal solutions for certain problems

  - but where they do, their speed is usually unbeatable

  ⇝ it is understanding where they succeed

(unknown quality)

- even where they are not optimal, greedy approaches can be efficient heuristics or approximation algorithms

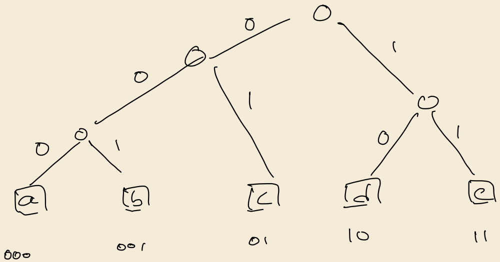
c-approximation = at most factor c worse than optimum

Plan for the Unit

- We will first see a few examples (known and new) of greedy algorithms to make the vague generic description concrete
  - in particular minimum spanning trees and shortest paths in graphs
- Unlike other algorithm design techniques, greedy algorithms have a formal basis: *matroids* (and *greedoids*)
  - The second part will introduce these and how they can unify correctness proofs

A First Example: Reunion With An Old Friend

- We have seen an example of a Greedy Algorithm in Unit 7: *Huffman Codes!*
- Recall the problem:
  - **Given:** Set of symbols $\Sigma = [0..\sigma)$, weights $w : \Sigma \to \mathbb{R}_{\geq 0}$
  - **Goal:** prefix code E (= code trie) that minimizes $\sum_{c\in\Sigma} w(c) \cdot |E(c)|$



A First Example: Reunion With An Old Friend

- We have seen an example of a Greedy Algorithm in Unit 7: *Huffman Codes!*
- Recall the problem:
  - **Given:** Set of symbols $\Sigma = [0..\sigma)$, weights $w : \Sigma \to \mathbb{R}_{\geq 0}$
  - **Goal:** prefix code E (= code trie) that minimizes $\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

$\rightsquigarrow$ Since only *code tries* are valid, all solutions consist in repeatedly merging tries (starting from single-leaf tries, until single trie left)

- each merge contributes the subtree's total weight to overall cost (since all leaves in merged tries move one level down / all codewords get one extra bit)
- **Huffman's Algorithm:** Always choose current cheapest merge.

A First Example: Reunion With An Old Friend

- We have seen an example of a Greedy Algorithm in Unit 7: *Huffman Codes!*
- Recall the problem:
  - **Given:** Set of symbols $\Sigma = [0..\sigma)$, weights $w : \Sigma \to \mathbb{R}_{\geq 0}$
  - **Goal:** prefix code E (= code trie) that minimizes $\sum_{c \in \Sigma} w(c) \cdot |E(c)|$

$\rightsquigarrow$ Since only *code tries* are valid, all solutions consist in repeatedly merging tries (starting from single-leaf tries, until single trie left)

- each merge contributes the subtree's total weight to overall cost (since all leaves in merged tries move one level down / all codewords get one extra bit)
- **Huffman's Algorithm:** Always choose current cheapest merge. w.r.t
- In the correctness proof, we had to show: There is always an optimal code trie where the two lowest-weight symbols are siblings.

This is typical: To show that Greedy is optimal, we need a structural insight into optimal solutions.

11.2 How Can Greed Succeed?

Greed For Change

The Change-Making Problem (a. k. a. Coin-Exchange Problem)

- **Given:** a set of integer denominations of coins $w_1 < w_2 < \cdots < w_k$ with $w_1 = 1$,
  target value $n \in \mathbb{N}_{\geq 1}$

  (we have sufficient supply of all coins . . .)
- **Goal:** "fewest coins to give change n", i. e.,
  multiplicities $c_1, \ldots, c_k \in \mathbb{N}_{\geq 0}$ with $\sum_{i=1}^{k} c_i \cdot w_i = n$ minimizing $\sum_{i=1}^{k} c_i$

Greed For Change

The Change-Making Problem (a. k. a. Coin-Exchange Problem)

- **Given:** a set of integer denominations of coins $w_1 < w_2 < \cdots < w_k$ with $w_1 = 1$,
  target value $n \in \mathbb{N}_{\geq 1}$ (we have sufficient supply of all coins . . .)
- **Goal:** "fewest coins to give change n", i. e.,
  multiplicities $c_1, \ldots, c_k \in \mathbb{N}_{\geq 0}$ with $\sum_{i=1}^{k} c_i \cdot w_i = n$ minimizing $\sum_{i=1}^{k} c_i$

For Euro coins, denominations are 1¢, 2¢, 5¢, 10¢, 20¢, 50¢, 1€, and 2€.

$$\text{formally: } \underset{w_1}{1}, \underset{w_2}{2}, \underset{w_3}{5}, \underset{w_4}{10}, \underset{w_5}{20}, \underset{w_6}{50}, \underset{w_7}{100}, \text{ and } \underset{w_8}{200}.$$

Greed For Change

The Change-Making Problem (a. k. a. Coin-Exchange Problem)

- **Given:** a set of integer denominations of coins $w_1 < w_2 < \cdots < w_k$ with $w_1 = 1$,
  target value $n \in \mathbb{N}_{\geq 1}$
  (we have sufficient supply of all coins . . .)
- **Goal:** "fewest coins to give change n", i. e.,
  multiplicities $c_1, \ldots, c_k \in \mathbb{N}_{\geq 0}$ with $\sum_{i=1}^{k} c_i \cdot w_i = n$ minimizing $\sum_{i=1}^{k} c_i$

For Euro coins, denominations are 1¢ , 2¢ , 5¢ , 10¢ , 20¢ , 50¢ , 1€ , and 2€ .

formally: $\underset{w_1}{1}$, $\underset{w_2}{2}$, $\underset{w_3}{5}$, $\underset{w_4}{10}$, $\underset{w_5}{20}$, $\underset{w_6}{50}$, $\underset{w_7}{100}$, and $\underset{w_8}{200}$.

$\rightsquigarrow$ Simple greedy algorithm:
largest coins first

- optimal time ($O(k)$ if coins sorted)
- is $\sum c_i$ minimal?

```
procedure greedyChange(w[1..k], n):
    // Assumes 1 = w[1] < w[2] < ··· < w[k]
    for i := k, k − 1, . . . , 1:
        c[i] := ⌊n / w[i]⌋
        n := n − c[i] · w[i]
    // Now n == 0
    return c[1..k]
```

Clicker Question

Does greedyChange give the optimal answer for the Euro coins change-making problem?

- A Always
- B Sometimes
- C Never

→ sli.do/cs566

Clicker Question

Does greedyChange give the optimal answer for the Euro coins change-making problem?

- A Always ✓
- B ~~Sometimes~~
- C ~~Never~~

→ sli.do/cs566

Optimality of Greedy Euro-Change

- **Theorem:** greedyChange computes an optimal $c[1..8]$ for $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.

Optimality of Greedy Euro-Change

- **Theorem:** greedyChange computes an optimal $c[1..8]$ for $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.
  - The greedy algorithm can be interpreted as picking one coin at a time, each time choosing the largest possible denomination $\hat{w}(n) = \max\{w[i] : w[i] \leq n\}$.
  - We prove by induction over n: Any optimal solution for n must contain $\hat{w}(n)$.
    - $n = 1$: can only use $\hat{w}(n) = 1$ ✓

Optimality of Greedy Euro-Change

- **Theorem:** greedyChange computes an optimal $c[1..8]$ for
  $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.
  - The greedy algorithm can be interpreted as picking one coin at a time, each time choosing the largest possible denomination $\hat{w}(n) = \max\{w[i] : w[i] \leq n\}$.
  - We prove by induction over n: Any optimal solution for n must contain $\hat{w}(n)$.
    - $n = 1$: can only use $\hat{w}(n) = 1$ ✓
    - $n \in [2..5)$: Assume we had a solution without 2¢ $\rightsquigarrow$ must be $n \times$ 1¢ with $n \geq 2$;
      $\rightsquigarrow$ we can make this strictly better by replacing 1¢ 1¢ by 2¢ $\lightning$

Optimality of Greedy Euro-Change

- **Theorem:** greedyChange computes an optimal $c[1..8]$ for $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.
  - The greedy algorithm can be interpreted as picking one coin at a time, each time choosing the largest possible denomination $\hat{w}(n) = \max\{w[i] : w[i] \leq n\}$.
  - We prove by induction over n: Any optimal solution for n must contain $\hat{w}(n)$.
    - $n = 1$: can only use $\hat{w}(n) = 1$ ✓
    - $n \in [2..5)$: Assume we had a solution without 2¢ $\rightsquigarrow$ must be $n \times$ 1¢ with $n \geq 2$;
      $\rightsquigarrow$ we can make this strictly better by replacing 1¢ 1¢ by 2¢ ϟ
    - $n \in [5..10)$: Assume solution without 5¢ summing to $n \geq 5$.
      The solution must fall into one of the following cases:
      (a) $\geq 3 \times$ 2¢ $\rightsquigarrow$ replacing 2¢ 2¢ 2¢ by 5¢ 1¢ strictly better ϟ
      (b) $\leq 1 \times$ 2¢ $\rightsquigarrow$ value $n - 2 \geq 3$ without 2¢ ϟ by IH
      (c) $2 \times$ 2¢ and $\geq 1 \times$ 1¢ $\rightsquigarrow$ 2¢ 2¢ 1¢ $\longrightarrow$ 5¢ strictly better ϟ
      (d) $2 \times$ 2¢, no 1¢ $\rightsquigarrow$ only obtain value $\leq 4 < n$ ϟ

Optimality of Greedy Euro-Change

- **Theorem:** greedyChange computes an optimal $c[1..8]$ for $w[1..8] = [1, 2, 5, 10, 20, 50, 100, 200]$ for every $n \in N_{\geq 1}$.
  - The greedy algorithm can be interpreted as picking one coin at a time, each time choosing the largest possible denomination $\hat{w}(n) = \max\{w[i] : w[i] \leq n\}$.
  - We prove by induction over n: Any optimal solution for n must contain $\hat{w}(n)$.
    - $n = 1$: can only use $\hat{w}(n) = 1$ ✓
    - $n \in [2..5)$: Assume we had a solution without 2¢ $\rightsquigarrow$ must be $n \times$ 1¢ with $n \geq 2$;
      $\rightsquigarrow$ we can make this strictly better by replacing 1¢ 1¢ by 2¢ ϟ
    - $n \in [5..10)$: Assume solution without 5¢ summing to $n \geq 5$.
      The solution must fall into one of the following cases:
      (a) $\geq 3 \times$ 2¢ $\rightsquigarrow$ replacing 2¢ 2¢ 2¢ by 5¢ 1¢ strictly better ϟ
      (b) $\leq 1 \times$ 2¢ $\rightsquigarrow$ value $n - 2 \geq 3$ without 2¢ ϟ by IH
      (c) $2 \times$ 2¢ and $\geq 1 \times$ 1¢ $\rightsquigarrow$ 2¢ 2¢ 1¢ $\rightarrow$ 5¢ strictly better ϟ
      (d) $2 \times$ 2¢, no 1¢ $\rightsquigarrow$ only obtain value $\leq 4 < n$ ϟ
    - $n \in [10, 20)$: Any solution without 10¢ contains
      (a) 5¢ 5¢ $\rightsquigarrow$ replace by 10¢; or
      (b) at most one 5¢ $\rightsquigarrow$ at least value 5 without 5¢ ϟ by IH

Optimality of Greedy Euro-Change [2]

- … proof continued
  - $n \in [20..50)$ Without 20¢, we must have
    (a) 10¢ 10¢ $\rightarrow$ 20¢ ϟ
    (b) at most one 10¢ $\rightsquigarrow$ value $n - 10 \geq 10$ without 10¢ ϟ by IH

Optimality of Greedy Euro-Change [2]

- ... proof continued
  - $n \in [20..50)$ Without 20¢, we must have
    (a) 10¢ 10¢ $\rightarrow$ 20¢ ϟ
    (b) at most one 10¢ $\rightsquigarrow$ value $n - 10 \geq 10$ without 10¢ ϟ by IH
  - $n \in [50..100)$ Without 50¢, we must have
    (a) $\geq 3 \times$ 20¢ $\rightsquigarrow$ 20¢ 20¢ 20¢ $\rightarrow$ 50¢ 10¢ ϟ
    (b) $\leq 1 \times$ 20¢ $\rightsquigarrow$ value $n - 20 \geq 30$ without 20¢ ϟ by IH
    (c) $2 \times$ 20¢ and $\geq 1 \times$ 10¢ $\rightsquigarrow$ 20¢ 20¢ 10¢ $\rightarrow$ 50¢ ϟ
    (d) $2 \times$ 20¢, no 10¢ $\rightsquigarrow$ value $n - 40 \geq 10$ without 10¢ ϟ by IH

Optimality of Greedy Euro-Change [2]

- … proof continued
  - $n \in [20..50)$ Without 20¢, we must have
    (a) 10¢ 10¢ $\longrightarrow$ 20¢ ϟ
    (b) at most one 10¢ $\rightsquigarrow$ value $n - 10 \geq 10$ without 10¢ ϟ by IH
  - $n \in [50..100)$ Without 50¢, we must have
    (a) $\geq 3 \times$ 20¢ $\rightsquigarrow$ 20¢ 20¢ 20¢ $\longrightarrow$ 50¢ 10¢ ϟ
    (b) $\leq 1 \times$ 20¢ $\rightsquigarrow$ value $n - 20 \geq 30$ without 20¢ ϟ by IH
    (c) $2 \times$ 20¢ and $\geq 1 \times$ 10¢ $\rightsquigarrow$ 20¢ 20¢ 10¢ $\longrightarrow$ 50¢ ϟ
    (d) $2 \times$ 20¢, no 10¢ $\rightsquigarrow$ value $n - 40 \geq 10$ without 10¢ ϟ by IH
  - $n \in [100..200)$: as for $n \in [10, 20)$, *mutatis mutandis*.
  - $n \geq 200$: as for $n \in [20, 50)$. $\square$

Optimality of Greedy Euro-Change [2]

- … proof continued
  - $n \in [20..50)$ Without 20¢, we must have
    (a) 10¢ 10¢ $\longrightarrow$ 20¢ ϟ
    (b) at most one 10¢ $\rightsquigarrow$ value $n - 10 \geq 10$ without 10¢ ϟ by IH
  - $n \in [50..100)$ Without 50¢, we must have
    (a) $\geq 3 \times$ 20¢ $\rightsquigarrow$ 20¢ 20¢ 20¢ $\longrightarrow$ 50¢ 10¢ ϟ
    (b) $\leq 1 \times$ 20¢ $\rightsquigarrow$ value $n - 20 \geq 30$ without 20¢ ϟ by IH
    (c) $2 \times$ 20¢ and $\geq 1 \times$ 10¢ $\rightsquigarrow$ 20¢ 20¢ 10¢ $\longrightarrow$ 50¢ ϟ
    (d) $2 \times$ 20¢, no 10¢ $\rightsquigarrow$ value $n - 40 \geq 10$ without 10¢ ϟ by IH
  - $n \in [100..200)$: as for $n \in [10, 20)$, *mutatis mutandis*.
  - $n \geq 200$: as for $n \in [20, 50)$.
- The same arguments work for adding coins $1 \cdot 10^m, 2 \cdot 10^m, 5 \cdot 10^m$ for $m = 3, 4, \ldots$

Optimality of Greedy Euro-Change [2]

- ... proof continued
  - $n \in [20..50)$ Without 20¢, we must have
    (a) 10¢ 10¢ $\rightarrow$ 20¢ ↯
    (b) at most one 10¢ $\rightsquigarrow$ value $n - 10 \geq 10$ without 10¢ ↯ by IH
  - $n \in [50..100)$ Without 50¢, we must have
    (a) $\geq 3 \times$ 20¢ $\rightsquigarrow$ 20¢ 20¢ 20¢ $\rightarrow$ 50¢ 10¢ ↯
    (b) $\leq 1 \times$ 20¢ $\rightsquigarrow$ value $n - 20 \geq 30$ without 20¢ ↯ by IH
    (c) $2 \times$ 20¢ and $\geq 1 \times$ 10¢ $\rightsquigarrow$ 20¢ 20¢ 10¢ $\rightarrow$ 50¢ ↯
    (d) $2 \times$ 20¢, no 10¢ $\rightsquigarrow$ value $n - 40 \geq 10$ without 10¢ ↯ by IH
  - $n \in [100..200)$: as for $n \in [10, 20)$, *mutatis mutandis*.
  - $n \geq 200$: as for $n \in [20, 50)$.
- The same arguments work for adding coins $1 \cdot 10^m, 2 \cdot 10^m, 5 \cdot 10^m$ for $m = 3, 4, \ldots$

That went smoothly!
And we proved a nice structural statement about how optimal solutions look like as a bonus.

Maybe Greedy always works?

Greed Can Mislead

- *Unfortunately, No.* See $w = (1, 3, 4)$ and $n = 6$.

4 1 1

3 3

Greed Can Mislead

- *Unfortunately, No.* See $w = (1, 3, 4)$ and $n = 6$.
  or $w = (1, 4, 9)$ and $n = 12$

Where/Why does our proof from above fail?

Greed Can Mislead

- *Unfortunately, No.* See $w = (1, 3, 4)$ and $n = 6$. *Where/Why does our proof from above fail?*
  or $w = (1, 4, 9)$ and $n = 12$
- Indeed, Greedy can be **arbitrarily bad** compared to the optimal solution:
  See $w = (1, 999, 1000)$ and $n = 1998$.

$\rightsquigarrow$ Need to be careful about the details of a correctness argument for greedy algorithms.

Greed Can Mislead

- *Unfortunately, No.* See $w = (1,3,4)$ and $n = 6$. *Where/Why does our proof from above fail?*
  or $w = (1,4,9)$ and $n = 12$
- Indeed, Greedy can be **arbitrarily bad** compared to the optimal solution:
  See $w = (1,999,1000)$ and $n = 1998$.

$\rightsquigarrow$ Need to be careful about the details of a correctness argument for greedy algorithms.

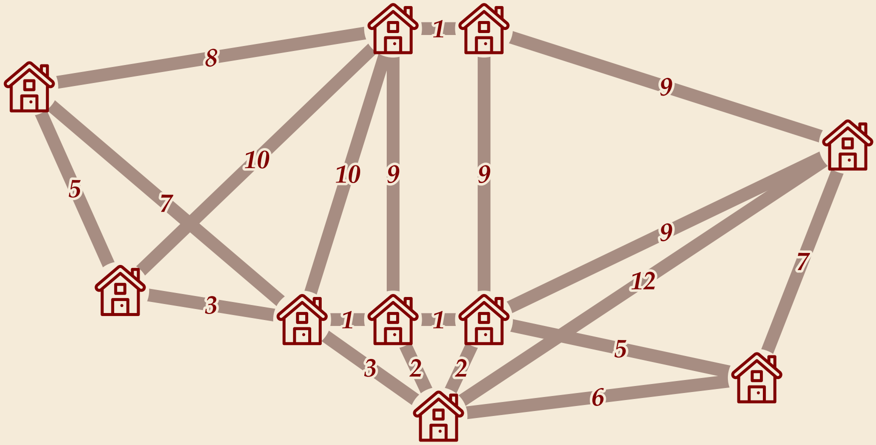
- The Change-Making problem is still only partially understood.
  - Exactly characterizing the denomination sequences that are optimally handled by greedyChange is an **open research problem**.
    - Sufficient criteria for "greed-compatible" denominations found in the literature.
  - The general problem is (weakly) NP-hard
  - Yet, for moderate n, we will see a solution for general denomination sequences later!

11.3 Greed in Graphs I: MSTs

Metaphor: Planning an electricity grid

Given: Houses to be connected to central power grid
Possible connections with building costs given

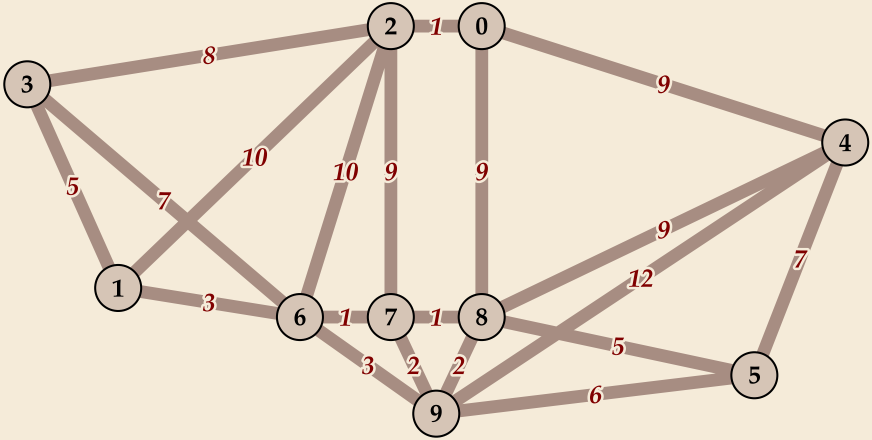
Goal: Cheapest way to get every house connected



Metaphor: Planning an electricity grid

Given: Houses to be connected to central power grid
Possible connections with building costs given

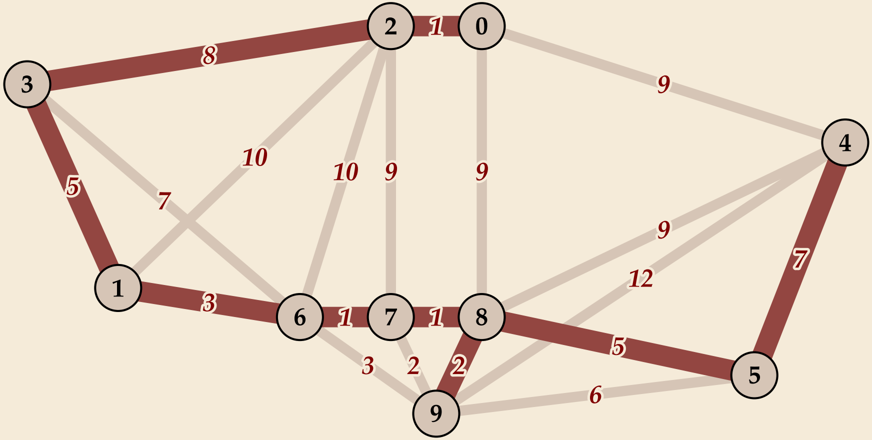
Goal: Cheapest way to get every house connected



Metaphor: Planning an electricity grid

Given: Houses to be connected to central power grid
Possible connections with building costs given

Goal: Cheapest way to get every house connected



Clicker Question

Which algorithm allows to efficiently test whether a given (undirected) graph is connected?

A bubble sort

B depth-first search

C breadth-first search

D generic tricolor search

E Kosaraju-Sharir's algorithm

F Dijkstra's algorithm

G Edmonds-Karp algorithm

→ sli.do/cs566

Clicker Question

Which algorithm allows to efficiently test whether a given (undirected) graph is connected?

A ~~bubble sort~~

B depth-first search ✓

C breadth-first search ✓

D generic tricolor search ✓

E Kosaraju-Sharir's algorithm ✓

F ~~Dijkstra's algorithm~~

G ~~Edmonds-Karp algorithm~~

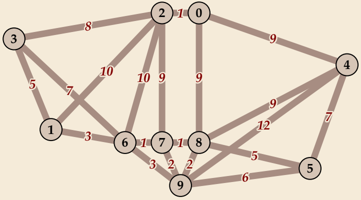
→ sli.do/cs566

The Minimum Spanning Tree (MST) Problem

Given: **un**directed, edge-**weighted**, simple,
connected graph $G = (V, E, c)$

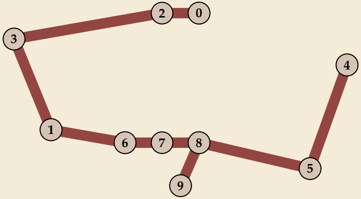
no self loops,
no parallel edges

Formally: Recall assumption $V = [0..n)$ ($\rightsquigarrow$ array indices)
edges $E \subseteq \big\{\{u, v\} : u, v \in V \wedge u \neq v\big\}$
edge weights (costs) $c : E \to \mathbb{R}_{\ge 0}$
for all $u, v \in V$ there exists a path $u \rightsquigarrow v$ in (V, E)



Goal: a **spanning tree** (V, T)
with **minimal** total cost $c(T) := \sum_{e \in T} c(e)$

Formally: $T \subseteq E$
(V, T) is connected and acyclic ("spanning tree")
for every spanning tree (V, T') of G we have $c(T') \ge c(T)$.



Further MST Applications

Direct Applictions

- single-linkage hierarchical clustering
- Bottleneck-shortest paths
- Approximation algorithms, e. g.,
  - Christofides's Metric TSP Approximation
  - Steiner-tree problem

As a cheap subroutine

- Routing protocols
- medical image processing
- ...

Interlude: On Varieties of Trees

We freely use "tree" to mean different things in different contexts . . . mind the confusion.

- here: "tree" = *undirected, nonplane tree* = an undirected, connected and acyclic graph

in spanning *tree*

no order on edges

Interlude: On Varieties of Trees

⚠ *We freely use "tree" to mean different things in different contexts . . . mind the confusion.*

- here: "tree" = *undirected, nonplane tree* = an undirected, connected and acyclic graph

  in spanning *tree*

  no order on edges

The digraph flavor is a rooted tree: (hence undirected trees sometimes called *unrooted*)

- *rooted (nonplane/unordered) tree* = **digraph** (V, E) with *root* $r \in V$ s.t.
  $\forall v \in V \setminus \{r\} : d_{\text{out}}(v) = 1$ and $d_{\text{out}}(r) = 0$

  out-degree = #outgoing edges

We draw trees with the single(!) root on top . . .

Interlude: On Varieties of Trees

We freely use "tree" to mean different things in different contexts ... mind the confusion.

- here: "tree" $=$ *undirected, nonplane tree* $=$ an undirected, connected and acyclic graph

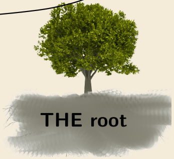
in spanning *tree*

no order on edges

The digraph flavor is a rooted tree: (hence undirected trees sometimes called *unrooted*)

- *rooted (nonplane/unordered) tree* $=$ **digraph** (V, E) with *root* $r \in V$ s.t.
  $\forall v \in V \setminus \{r\} : d_{\text{out}}(v) = 1$ and $d_{\text{out}}(r) = 0$

out-degree = #outgoing edges



We draw trees with the single(!) root on top ...

ordered rooted

Interlude: On Varieties of Trees

⚠ *We freely use "tree" to mean different things in different contexts . . . mind the confusion.*

- here: "tree" $=$ *undirected, nonplane tree* $=$ an undirected, connected and acyclic graph

  for this problem w/ [illegible]

  worst-case Θ-class

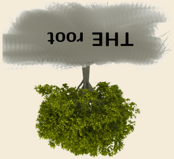
  in spanning *tree*

  no order on edges

The digraph flavor is a rooted tree: (hence undirected trees sometimes called *unrooted*)

- *rooted (nonplane/unordered) tree* $=$ **digraph** (V, E) with *root* $r \in V$ s.t.
  $\forall v \in V \setminus \{r\} : d_{\text{out}}(v) = 1$ and $d_{\text{out}}(r) = 0$

  out-degree = #outgoing edges



We draw trees with the single(!) root on top . . .

Interlude: On Varieties of Trees

⚠ *We freely use "tree" to mean different things in different contexts . . . mind the confusion.*

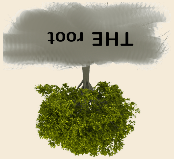
- here: "tree" = *undirected, nonplane tree* = an undirected, connected and acyclic graph

  in spanning *tree* no order on edges

The digraph flavor is a rooted tree: (hence undirected trees sometimes called *unrooted*)

- *rooted (nonplane/unordered) tree* = **digraph** (V, E) with *root* $r \in V$ s.t.
  $\forall v \in V \setminus \{r\} : d_{\text{out}}(v) = 1$ and $d_{\text{out}}(r) = 0$

  out-degree = #outgoing edges



We draw trees with the single(!) root on top . . .

Other "trees" don't originate from graphs naturally, but rather from recursion / terms:

- *binary tree* = a null pointer or a node with left and right children, each a binary tree

  (formally: the set of binary trees is the smallest fixed point of that construction)

- *ordinal trees* = a node with a sequence of 0 or more children, each ordinal trees
  = rooted ordered trees (rooted unordered + total order on children)

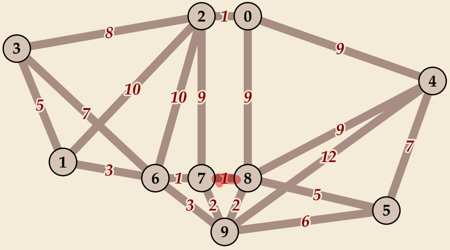
- plus many more variants out there . . . $\rightsquigarrow$ if in doubt, double check definitions!

A Naive Approach

How to start finding an MST?

Using the **cheapest** edge shouldn't hurt ...

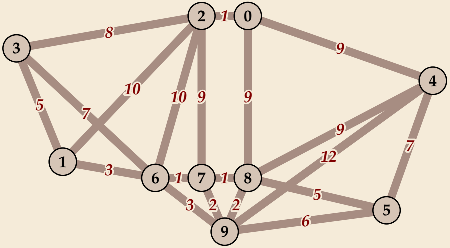
procedure greedyMST(V, E, c):
  // Assume (V, E) *is simple & connected,* $c : E \to \mathbb{R}_{\geq 0}$
  $T := \emptyset$
  while (V, T) not connected
    e := cheapest edge that doesn't close a cycle in T
    $T := T \cup \{e\}$
  return T



A Naive Approach Works – Kruskal's Algorithm

How to start finding an MST?

Using the **cheapest** edge shouldn't hurt . . .



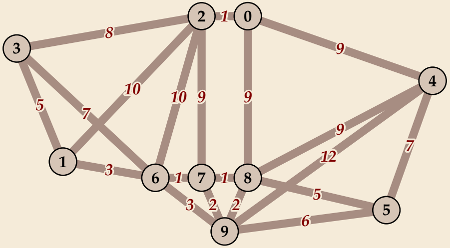

```
1  procedure kruskalMST(V, E, c):
2      // Assume (V, E) is simple & connected, c : E → ℝ≥0
3      T := ∅
4      while (V, T) not connected
5          e := cheapest edge that doesn't close a cycle in T
6          T := T ∪ {e}
7      return T
```

Apart from implementing line 4 and line 5 efficiently, this *is* **Kruskal's Algorithm**!

A Naive Approach Works – Kruskal's Algorithm

How to start finding an MST?

Using the **cheapest** edge shouldn't hurt . . .



---

1 **procedure** kruskalMST(V, E, c):

2   *// Assume* (V, E) *is simple & connected,* $c : E \to \mathbb{R}_{\geq 0}$

3     $T := \emptyset$

4   **while** (V, T) not connected

5   e := cheapest edge that doesn't close a cycle in T

6         $T := T \cup \{e\}$

7   **return** T

Apart from implementing line 4 and line 5 efficiently, this *is* **Kruskal's Algorithm**!

As so often with greedy algorithms, the hardest bit is the correctness argument. We have:

Theorem: Kruskal's Algorithm finds a minimum spanning tree.
This immediately follows from proving the following invariant:

Kruskal's Invariant: There is some MST T^* with $T \subseteq T^*$.

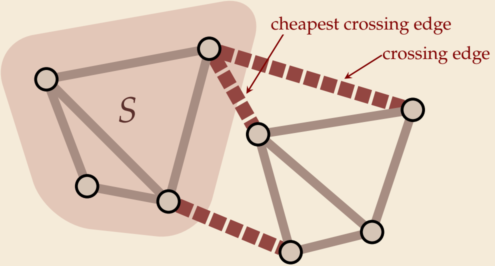
henceforth: identify MST with its edge set

Crossing Edges and the MST-Cut Lemma

To prove the correctness of Kruskal's Algorithm, we need a tool.

Notation:

- **Cut** S:
  non-trivial set of vertices $\emptyset \neq S \subsetneq V$
- **crossing edge** e wrt. cut S:
  $e = \{u, v\}$ with $u \in S, v \in \bar{S} := V \setminus S$

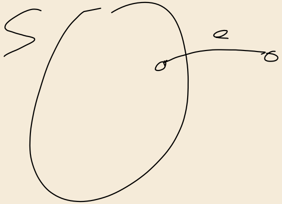


The MST-Cut Lemma:
Let T^* be an MST und $W \subseteq T^*$.
For every cut S, not cutting any edges in W, and
every *cheapest* crossing edge e wrt. S
there is an MST $\hat{T}^*$ that contains $W \cup \{e\}$.

Proof of MST-Cut Lemma

Proof:

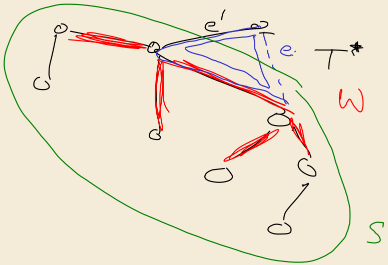
- Case 1: $e \in T^*$.
  Then picking $\hat{T}^* = T^*$ proves the claim.



Proof of MST-Cut Lemma

Proof:

- Case 1: $e \in T^*$.
  Then picking $\hat{T}^* = T^*$ proves the claim.
- Case 2: $e \notin T^*$.
  - $\rightsquigarrow$ $T^* \cup \{e\}$ contains unique cycle C using e.
  - Since e crosses cut S, C crosses S
  - $\rightsquigarrow$ There is a second crossing edge $e' \in C$.

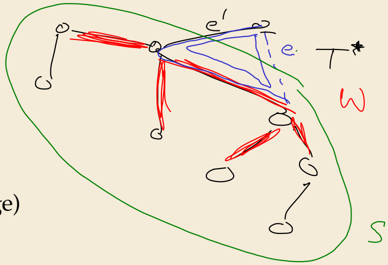


Proof of MST-Cut Lemma

The MST-Cut Lemma:
Let T^* be an MST und $W \subseteq T^*$.
For every cut S, not cutting any edges in W, and
every *cheapest* crossing edge e wrt. S
there is an MST $\hat{T}^*$ that contains $W \cup \{e\}$.

Proof:

- Case 1: $e \in T^*$.
  Then picking $\hat{T}^* = T^*$ proves the claim.
- Case 2: $e \notin T^*$.
  - $\rightsquigarrow$ $T^* \cup \{e\}$ contains unique cycle C using e.
  - Since e crosses cut S, C crosses S
  - $\rightsquigarrow$ There is a second crossing edge $e' \in C$.
  - Since e' is crossing, $e' \notin W$
  - by assumption, $c(e) \le c(e')$ (we pick cheapest crossing edge)
  - $\rightsquigarrow$ $\hat{T}^* = T^* \cup \{e\} \setminus \{e'\}$ is a spanning tree, and $W \cup \{e\} \subseteq \hat{T}^*$
  - $c(\hat{T}^*) = c(T^*) + c(e) - c(e') \le c(T^*)$
  - $\rightsquigarrow$ $\hat{T}^*$ is an **MST**.



Kruskal's Algorithm – Correctness

With these preparations, we can prove

$\boxed{\textbf{Kruskal's Invariant: } \text{There is some MST } T^* \text{ with } T \subseteq T^*.}$

Proof: by induction over the loop iterations

- IB: initially $T = \emptyset$ and $\emptyset \subseteq T^*$ for every MST T^*.
- IH: Assume the invariant is after the ith iteration.

Kruskal's Algorithm – Correctness

With these preparations, we can prove

$\boxed{\textbf{Kruskal's Invariant:}\ \text{There is some MST } T^* \text{ with } T \subseteq T^*.}$

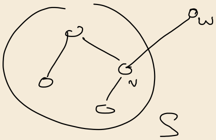
Proof: by induction over the loop iterations

- IB: initially $T = \emptyset$ and $\emptyset \subseteq T^*$ for every MST T^*.
- IH: Assume the invariant is after the ith iteration.
- IS: Let $e = vw$ be the edge considered in iteration $i + 1$.
  - Let S be the connected component of v in (V, T) (T: before potentially adding e)
  - Case 1: $w \in S$.
    Then e closes a cycle in T and is not added to T.
    $\rightsquigarrow$ invariant still satisfied.

Kruskal's Algorithm – Correctness

With these preparations, we can prove

Kruskal's Invariant: There is some MST T^* with $T \subseteq T^*$.



Proof: by induction over the loop iterations

- IB: initially $T = \emptyset$ and $\emptyset \subseteq T^*$ for every MST T^*.
- IH: Assume the invariant is after the ith iteration.
- IS: Let $e = vw$ be the edge considered in iteration $i + 1$.
  - Let S be the connected component of v in (V, T) (T: before potentially adding e)
  - Case 1: $w \in S$.
    Then e closes a cycle in T and is not added to T.
    $\leadsto$ invariant still satisfied.
  - Case 2: $w \notin S$.
    Then e is a crossing edge wrt. S; must be a cheapest crossing edge by choice of e.
    $\leadsto$ by inv. $\exists$ MST $T^* \supseteq T$ and by MST-Cut Lemma, there is an MST $\hat{T}^* \supseteq T \cup \{e\}$
    $\leadsto$ Invariant still satisfied.

Kruskal's Algorithm – Correctness

With these preparations, we can prove

Kruskal's Invariant: There is some MST T^* with $T \subseteq T^*$.

Proof: by induction over the loop iterations

- IB: initially $T = \emptyset$ and $\emptyset \subseteq T^*$ for every MST T^*.
- IH: Assume the invariant is after the ith iteration.
- IS: Let $e = vw$ be the edge considered in iteration $i+1$.
  - Let S be the connected component of v in (V, T) (T: before potentially adding e)
  - Case 1: $w \in S$.
    Then e closes a cycle in T and is not added to T.
    $\rightsquigarrow$ invariant still satisfied.
  - Case 2: $w \notin S$.
    Then e is a crossing edge wrt. S; must be a cheapest crossing edge by choice of e.
    $\rightsquigarrow$ by inv. $\exists$ MST $T^* \supseteq T$ and by MST-Cut Lemma, there is an MST $\hat{T}^* \supseteq T \cup \{e\}$
    $\rightsquigarrow$ Invariant still satisfied.

The MST-Cut Lemma:
Let T^* be an MST und $W \subseteq T^*$.
For every cut S, not cutting any edges in W, and
every *cheapest* crossing edge e wrt. S
there is an MST $\hat{T}^*$ that contains $W \cup \{e\}$.

$W = T$

S

Since we only terminate when T is spanning, upon termination $T = T^*$ for an MST T^*.

Kruskal's Algorithm – Data Structures

For an efficient implementation of Kruskal's algorithm, we need to efficiently

1. check whether T is spanning
2. find the next cheapest edge to consider
3. test whether an edge closes a cycle

Kruskal's Algorithm – Data Structures

For an efficient implementation of Kruskal's algorithm, we need to efficiently

1. check whether T is spanning
2. find the next cheapest edge to consider
3. test whether an edge closes a cycle

Each can be supported as follows:

1. Since we maintain T acyclic, checking $|T| = n - 1$ suffices!
2. It suffices to pre-sort E by weight!
   - We only ever grow T, so if e is closing a cycle now, it will for good.

   $\rightsquigarrow$ Once discarded, an edge need not be looked at ever again.

Kruskal's Algorithm – Data Structures

For an efficient implementation of Kruskal's algorithm, we need to efficiently

1. check whether T is spanning
2. find the next cheapest edge to consider
3. test whether an edge closes a cycle

Each can be supported as follows:

1. Since we maintain T acyclic, checking $|T| = n - 1$ suffices!
2. It suffices to pre-sort E by weight!
   - We only ever grow T, so if e is closing a cycle now, it will for good.
   - $\rightsquigarrow$ Once discarded, an edge need not be looked at ever again.
3. Use a **Union-Find data structure** (see Algorithmen & Datenstrukturen!)
   - dynamically maintain connected components
   - initially, every vertex has its own id
   - adding vw to T $\rightsquigarrow$ call `union`(v, w)
   - vw closes a cycle *iff* `find`(v) `==` `find`(w)

∉ exam

Kruskal's Algorithm – Data Structures

For an efficient implementation of Kruskal's algorithm, we need to efficiently

1. check whether T is spanning
2. find the next cheapest edge to consider
3. test whether an edge closes a cycle

Each can be supported as follows:

1. Since we maintain T acyclic, checking $|T| = n - 1$ suffices!
2. It suffices to pre-sort E by weight!
   - We only ever grow T, so if e is closing a cycle now, it will for good.

   $\rightsquigarrow$ Once discarded, an edge need not be looked at ever again.
3. Use a **Union-Find data structure** (see Algorithmen & Datenstrukturen!)
   - dynamically maintain connected components
   - initially, every vertex has its own id
   - adding vw to T $\rightsquigarrow$ call `union`(v,w)
   - vw closes a cycle *iff* `find`(v) `==` `find`(w)

$\rightsquigarrow$ $O(m \log m) = O(m \log n)$ time and $O(m)$ extra space.

Clicker Question

What is the running time of Prim's algorithm?

- A $\Theta(\log(n+m))$
- B $\Theta(n\sqrt{m})$
- C $\Theta(n+m)$
- D $\Theta(n^2+m)$
- E $\Theta(m+n\log n)$
- F $\Theta(n+m\log n)$
- G $\Theta(m\log n)$
- H $\Theta(m\log m)$
- I $\Theta(n\log(n+m))$
- J $\Theta(m^2)$

→ sli.do/cs566

Clicker Question

What is the running time of Prim's algorithm?

A $\cancel{\Theta(\log(n+m))}$

B $\cancel{\Theta(n\sqrt{m})}$

C $\cancel{\Theta(n+m)}$

D $\cancel{\Theta(n^2+m)}$

E $\Theta(m + n\log n)$ ✓

F $\cancel{\Theta(n + m\log n)}$

G $\Theta(m\log n)$ ✓

H $\Theta(m\log m)$ ✓

I $\cancel{\Theta(n\log(n+m))}$

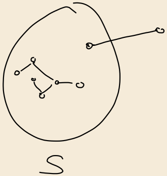
J $\cancel{\Theta(m^2)}$

→ sli.do/cs566

11.4 Greed in Graphs II: Prim's MST Algorithm

Prim's Algorithm

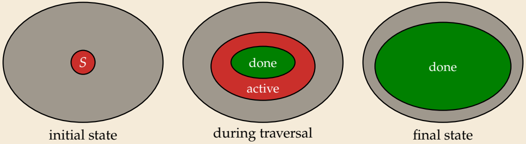
- An alternative greedy approach that tries to consider only crossing edges.
  - start with $S = \{s\}$ for some vertex s
  - only consider edges vw for some $v \in S, w \notin S$ (crossing edges)
  - add cheapest crossing edge vw to T and add w to S
  - repeat until $|T| = n - 1$
  - $\rightsquigarrow$ T invariably a single tree



Prim's Algorithm

- An alternative greedy approach that tries to consider only crossing edges.
  - start with $S = \{s\}$ for some vertex s
  - only consider edges vw for some $v \in S$, $w \notin S$ (crossing edges)
  - add cheapest crossing edge vw to T and add w to S
  - repeat until $|T| = n - 1$
  - $\rightsquigarrow$ T invariably a single tree

$\rightsquigarrow$ a graph traversal with tree edges T!

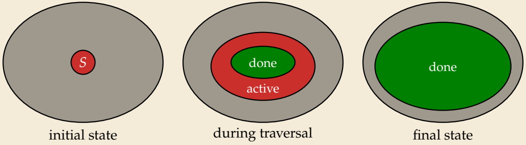


The MST-Cut Lemma:
Let T^* be an MST und $W \subseteq T^*$.
For every cut S, not cutting any edges in W, and
every *cheapest* crossing edge e wrt. S
there is an MST $\hat{T}^*$ that contains $W \cup \{e\}$.

Prim's Algorithm

- An alternative greedy approach that tries to consider only crossing edges.
  - start with $S = \{s\}$ for some vertex s
  - only consider edges vw for some $v \in S$, $w \notin S$ (crossing edges)
  - add cheapest crossing edge vw to T and add w to S
  - repeat until $|T| = n - 1$
  - $\rightsquigarrow$ T invariably a single tree

$\rightsquigarrow$ a graph traversal with tree edges T!



The MST-Cut Lemma:
Let T^* be an MST und $W \subseteq T^*$.
For every cut S, not cutting any edges in W, and
every *cheapest* crossing edge e wrt. S
there is an MST $\hat{T}^*$ that contains $W \cup \{e\}$.

$\rightsquigarrow$ Correctness as for Kruskal's algorithm: $\boxed{\textbf{Invariant: } \exists \text{ MST } T^* \text{ with } T \subseteq T^*.}$

IB: initially true with $T = \emptyset$

IS: whenever we add an edge, it is the cheapest crossing edge w.r.t. cut $(S, \bar{S})$.

Prim's Algorithm – Lazy Implementation

How to efficiently find the cheapest crossing edge?

- **Option 1:** Maintain priority queue Q of **edges**, ordered by weight.

```
procedure lazyPrimMST(G):
    // Assume G = (V, E, c) simple & connected, c : E → ℝ≥0
    T := ∅;  inS[0..n) := false
    Q := new MinPQ()
    visit(0)
    while |T| < n − 1:
        vw := Q.delMin()
        if ¬inS[w] then visit(w); T.insert(vw) end if
        if ¬inS[v] then visit(v); T.insert(wv) end if
    return T

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w] then Q.insert(vw, c) // w now active
    inS[v] := true // v now done
```

- Lazy Prim: check if vw is crossing *lazily*
  i. e., only after `delMin`

Prim's Algorithm – Lazy Implementation

How to efficiently find the cheapest crossing edge?

- **Option 1**: Maintain priority queue Q of **edges**, ordered by weight.

```
procedure lazyPrimMST(G):
    // Assume G = (V, E, c) simple & connected, c : E → ℝ≥0
    T := ∅;  inS[0..n) := false
    Q := new MinPQ()
    visit(0)
    while |T| < n − 1:
        vw := Q.delMin()
        if ¬inS[w] then visit(w); T.insert(vw) end if
        if ¬inS[v] then visit(v); T.insert(wv) end if
    return T

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w] then Q.insert(vw, c) // w now active
    inS[v] := true // v now done
```

- Lazy Prim: check if vw is crossing *lazily*
  i. e., only after `delMin`
- An instance of tricolor graph traversal
  - $v \in$ ***done*** iff $inS[v]$
  - all edges *to* ***active*** vertices are in Q
  - $\rightsquigarrow$ visit every edge at most once
- size of Q always $\leq m$ $\rightsquigarrow$ **space** $O(m)$

Prim's Algorithm – Lazy Implementation

How to efficiently find the cheapest crossing edge?

- **Option 1:** Maintain priority queue Q of **edges**, ordered by weight.

```
procedure lazyPrimMST(G):
    // Assume G = (V, E, c) simple & connected, c : E → ℝ≥0
    T := ∅;  inS[0..n) := false
    Q := new MinPQ()
    visit(0)
    while |T| < n − 1:
        vw := Q.delMin()
        if ¬inS[w] then visit(w); T.insert(vw) end if
        if ¬inS[v] then visit(v); T.insert(wv) end if
    return T

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w] then Q.insert(vw, c) // w now active
    inS[v] := true // v now done
```

- Lazy Prim: check if vw is crossing *lazily* i. e., only after `delMin`
- An instance of tricolor graph traversal ↳ $O(n+m)$ contribution
  - $v \in$ ***done*** iff $inS[v]$
  - all edges *to* ***active*** vertices are in Q
  - ⇝ visit every edge at most once
- size of Q always $\le m$ ⇝ **space** $O(m)$
- **Running time:** $(n \le m+1)$
  - need m calls to `insert` and $n-1$ `delMin`s
  - ⇝ with binary heaps, total time $O(m \log m) = O(m \log n)$

$m \le n^2$

Prim's Algorithm – Lazy Implementation

How to efficiently find the cheapest crossing edge?

- **Option 1:** Maintain priority queue Q of **edges**, ordered by weight.

```
procedure lazyPrimMST(G):
    // Assume G = (V, E, c) simple & connected, c : E → ℝ≥0
    T := ∅;  inS[0..n) := false
    Q := new MinPQ()
    visit(0)
    while |T| < n − 1:
        vw := Q.delMin()
        if ¬inS[w] then visit(w); T.insert(vw) end if
        if ¬inS[v] then visit(v); T.insert(wv) end if
    return T

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w] then Q.insert(vw, c) // w now active
    inS[v] := true // v now done
```

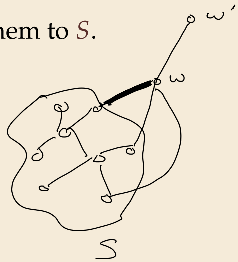
- Lazy Prim: check if vw is crossing *lazily*
  i. e., only after `delMin`
- An instance of tricolor graph traversal
  - $v \in$ ***done*** iff $inS[v]$
  - all edges *to* ***active*** vertices are in Q
  - ⇝ visit every edge at most once
- size of Q always $\le m$ ⇝ **space** $O(m)$
- **Running time:**
  - need m calls to `insert`
    and $n - 1$ `delMin`s
  - ⇝ with binary heaps, total time
    $O(m \log m) = O(m \log n)$
  - with Fibonacci heaps even
    $O(m + n \log n)$ (insert amortized $O(1)$ time)

Easy modification: store parent in tree rooted at vertex 0

Prim's Algorithm – Eager Implementation

We can reduce the extra space to $O(n)$ if we avoid storing multiple edges to the same $w \in \bar{S}$.

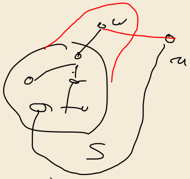
- **Option 2:** Maintain priority queue Q of **vertices** in $\bar{S}$, ordered by **weight of cheapest edge** connecting them to S.
  - call that weight the *distance*, $dist[w]$, of $w \in \bar{S}$ from S.
    ($dist[w] = 0$ if $w \in S$, $dist[w] = \infty$ if no single edge to S)



Prim's Algorithm – Eager Implementation

We can reduce the extra space to $O(n)$ if we avoid storing multiple edges to the same $w \in \bar{S}$.

- **Option 2:** Maintain priority queue Q of **vertices** in $\bar{S}$, ordered by **weight of cheapest edge** connecting them to S.



  - call that weight the *distance*, $dist[w]$, of $w \in \bar{S}$ from S.
    ($dist[w] = 0$ if $w \in S$, $dist[w] = \infty$ if no single edge to S)
  - after adding a vertex u to S, distance to w can **shrink** (to $c(uw)$) (but never grow)

  ⇝ need a MinPQ that supports `decreaseKey`

    - implementation hassle: efficient implementations require a "pointer" into data structure
      cleaner design: let data structure handle pointers internally

Prim's Algorithm – Eager Implementation

We can reduce the extra space to $O(n)$ if we avoid storing multiple edges to the same $w \in \bar{S}$.

- **Option 2:** Maintain priority queue Q of **vertices** in $\bar{S}$,
  ordered by **weight of cheapest edge** connecting them to S.
  - call that weight the *distance*, $dist[w]$, of $w \in \bar{S}$ from S.
    ($dist[w] = 0$ if $w \in S$, $dist[w] = \infty$ if no single edge to S)
  - after adding a vertex u to S, distance to w can **shrink** (to $c(uw)$) (but never grow)
  - ⇝ need a MinPQ that supports `decreaseKey`
    - implementation hassle: efficient implementations require a "pointer" into data structure
      cleaner design: let data structure handle pointers internally

⇝ **IndexMinPQ**:

(use ST otherwise) (use amortized doubling otherwise)

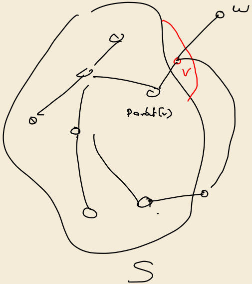
- **Assumption:** stored objects are from $[0..n)$ **and** n known/fixed at construction time
- IndexMinPQ implementations maintain array positions
  e. g., for binary heaps, maintain $heapIndex[0..n)$, update whenever heap modified
- ⇝ easy to support `decreaseKey`(i, p') and `contains`(i)
  (for a full implementation see *Sedgewick & Wayne* or *Nebel & Wild*)

Prim's Algorithm – Eager Implementation Code

```
procedure primMST(G):
    // Assume G = (V, E, c) is simple & connected, c : E → ℝ≥0
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := ∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0)
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return {{father[v], v} : v ∈ [1..n)}

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w]
            if c < dist[w] // vw currently cheapest edge to w
                father[w] := v;  dist[w] := c
                if Q.contains(w) // w already active
                    Q.decreaseKey(w, c)
                else // w now becoming active
                    Q.insert(w, c)
                end if
            end if
        end for
    inS[v] := true;  dist[v] := 0 // v now done
```

- Eager Prim: filter edges eagerly!
  $\rightsquigarrow$ keep only **cheapest edge** to $w \in \bar{S}$ (namely $\{father[w], w\}$)



Prim's Algorithm – Eager Implementation Code

```
procedure primMST(G):
    // Assume G = (V, E, c) is simple & connected, c : E → ℝ≥0
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := ∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0)
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return {{father[v], v} : v ∈ [1..n)}

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w]
            if c < dist[w] // vw currently cheapest edge to w
                father[w] := v;  dist[w] := c
                if Q.contains(w) // w already active
                    Q.decreaseKey(w, c)
                else // w now becoming active
                    Q.insert(w, c)
            end if
        end if
    end for
    inS[v] := true;  dist[v] := 0 // v now done
```

- Eager Prim: filter edges eagerly!
  $\rightsquigarrow$ keep only **cheapest edge** to $w \in \bar{S}$ (namely $\{father[w], w\}$)
- Prototypical tricolor traversal variant
  - $v \in$ ***done*** iff $inS[v]$ == true
  - $v \in$ ***active*** iff Q.contains(v)
  - choose next vertex using PQ Q, iterative over its edges
- size of Q always $\le n$ $\rightsquigarrow$ **space** $O(n)$

Prim's Algorithm – Eager Implementation Code

```
procedure primMST(G):
    // Assume G = (V, E, c) is simple & connected, c : E → ℝ≥0
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := ∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0)
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return {{father[v], v} : v ∈ [1..n)}

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c
        if ¬inS[w]
            if c < dist[w] // vw currently cheapest edge to w
                father[w] := v;  dist[w] := c
                if Q.contains(w) // w already active
                    Q.decreaseKey(w, c)
                else // w now becoming active
                    Q.insert(w, c)
                end if
        end if
    end for
    inS[v] := true;  dist[v] := 0 // v now done
```

- Eager Prim: filter edges eagerly!
  $\leadsto$ keep only **cheapest edge** to $w \in \bar{S}$
  (namely $\{father[w], w\}$)
- Prototypical tricolor traversal variant
  - $v \in$ ***done*** iff $inS[v]$
  - $v \in$ ***active*** iff Q.contains(v)
  - choose next vertex using PQ Q, iterative over its edges
- size of Q always $\le n$ $\leadsto$ **space** $O(n)$
- **Running time:**
  - $n \times$ `insert`, $(n-1) \times$ `delMin`, up to $m \times$ `decreaseKey`

  $\leadsto$ with binary heaps $O(m \log n)$
  with Fibonacci heaps $O(m + n \log n)$

Minimum Spanning Trees – Discussion

- MSTs are a versatile modeling tool
- very efficient to compute even for arbitrary weights
- Prim's Algorithm (eager version) give best time and space and is efficient in practice

lower bound $\Omega(n+m)$ to "see" entire graph

Minimum Spanning Trees – Discussion

- 👍 MSTs are a versatile modeling tool
- 👍 very efficient to compute even for arbitrary weights
- 👍 Prim's Algorithm (eager version) give best time and space and is efficient in practice

*Above algorithms are **almost linear-time**, but not quite . . . can we find MSTs in linear time?*

Minimum Spanning Trees – Discussion

- MSTs are a versatile modeling tool
- very efficient to compute even for arbitrary weights
- Prim's Algorithm (eager version) give best time and space and is efficient in practice

Above algorithms are ***almost linear-time****, but not quite . . . can we find MSTs in linear time?*

- Yes, if graph is **dense**, e. g., $m = \Omega(n \log n)$. Then $O(m + n \log n) = O(m)$
  - stronger results known, as well
- Yes, for **integer** weights on the word-RAM (Fredman, Willard 1994)
- Yes, if **randomization** is allowed (Karger, Klein, Tarjan 1995)
  - uses that linear time suffices to *verify* a given ST as minimal(!)

Minimum Spanning Trees – Discussion

- MSTs are a versatile modeling tool
- very efficient to compute even for arbitrary weights
- Prim's Algorithm (eager version) give best time and space and is efficient in practice

Above algorithms are ***almost linear-time****, but not quite . . . can we find MSTs in linear time?*

- Yes, if graph is **dense**, e. g., $m = \Omega(n \log n)$. Then $O(m + n \log n) = O(m)$
  - stronger results known, as well
- Yes, for **integer** weights on the word-RAM (Fredman, Willard 1994)
- Yes, if **randomization** is allowed (Karger, Klein, Tarjan 1995)
  - uses that linear time suffices to *verify* a given ST as minimal(!)
- General (deterministic, comparison-based, on sparse graphs)? **Open research problem!**
  - Best known general time $O(m\alpha(m, n))$ where α is an "inverse Ackermann function"

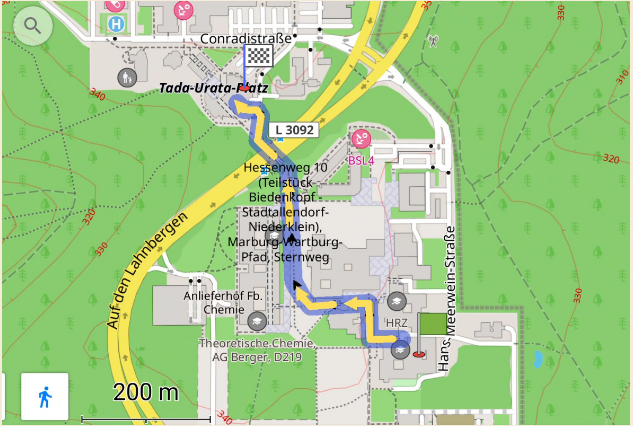
$\alpha(m, n) = \min\{z \ge 1 : A(z, 4\lceil m/n \rceil) > \lg n\}$
$A(0, x) = 2x,\ A(i, 0) = 0,\ A(i, 1) = 2,\ (i \ge 1),$
$A(i, x) = A\big(i-1, A(i, x-1)\big);\ (i \ge 1, x \ge 2)$

11.5 Greed in Graphs III: Shortest Paths

Metaphor: Route Planning

Given: Road network (map), current location, target location
crossings = vertices, roads = edges, road length = edge weight

Goal: Find shortest path from current location to target



SSSPP

It turns out that a cleaner algorithmic problem is to find shortest paths to *all* vertices.

Single Source Shortest Path Problem (SSSPP)

- **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
  with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
- **Goal:** a data structure that reports for every $v \in V$:
  $\delta_G(s, v)$: the shortest-path distance from s to v
  $\texttt{spath}(v)$: a shortest path from s to v (if it exists)

SSSPP

It turns out that a cleaner algorithmic problem is to find shortest paths to *all* vertices.

Single Source Shortest Path Problem (SSSPP)

- **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
  with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
- **Goal:** a data structure that reports for every $v \in V$:
  $\delta_G(s, v)$: the shortest-path distance from s to v
  $\texttt{spath}(v)$: a shortest path from s to v (if it exists)

Formally:

- for a walk $w[0..m]$ in G, we define $c(w) = \sum_{i=0}^{m-1} c\big(w[i]w[i+1]\big)$

SSSPP

It turns out that a cleaner algorithmic problem is to find shortest paths to *all* vertices.

Single Source Shortest Path Problem (SSSPP)

- **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$
  with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
- **Goal:** a data structure that reports for every $v \in V$:
  $\delta_G(s, v)$: the shortest-path distance from s to v
  $\texttt{spath}(v)$: a shortest path from s to v (if it exists)

Formally:

- for a walk $w[0..m]$ in G, we define $c(w) = \sum_{i=0}^{m-1} c\big(w[i]w[i+1]\big)$
- $\delta_G(s, v) \;=\; \inf\Big(\{+\infty\} \cup \big\{c(w) \;:\; w = w[0..m] \text{ a walk in } G \text{ with } w[0] = s \wedge w[m] = v\big\}\Big)$
  - Note: δ_G defined via all *s-v-walks*, not only *s-v-paths* (= vertex-single walks)
  - But we will see: In relevant scenarios, we can restrict to paths (hence the name)

SSSPP

It turns out that a cleaner algorithmic problem is to find shortest paths to *all* vertices.

Single Source Shortest Path Problem (SSSPP)

- **Given:** directed, edge-weighted, simple graph $G = (V, E, c)$ with edge costs $c : E \to \mathbb{R}$, a start vertex $s \in V$
- **Goal:** a data structure that reports for every $v \in V$:
  $\delta_G(s, v)$: the shortest-path distance from s to v
  $\texttt{spath}(v)$: a shortest path from s to v (if it exists)

Formally:

- for a walk $w[0..m]$ in G, we define $c(w) = \sum_{i=0}^{m-1} c\big(w[i]w[i+1]\big)$
- $\delta_G(s, v) \;=\; \inf\Big(\{+\infty\} \cup \big\{c(w) \;:\; w = w[0..m] \text{ a walk in } G \text{ with } w[0] = s \wedge w[m] = v\big\}\Big)$
  - Note: δ_G defined via all *s-v-walks*, not only *s-v-paths* (= vertex-single walks)
  - But we will see: In relevant scenarios, we can restrict to paths (hence the name)
- $\texttt{spath}(v)$ returns a walk w with $c(w) = \delta_G(s, v)$ if such a walk exists

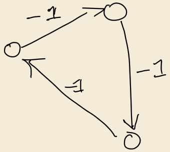
The Trouble with Negative Cycles

- The complications in the definition all stem from **negative-weight edges**

$$\delta_G(s,v) \;=\; \boxed{\inf\left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\right)}$$

- In general, $\delta_G(s,v)$ can be
  - $+\infty$ if there is no s-v-walk at all, or (“no-path case” easy to detect and handle)
  - $-\infty$ if there are s-v-*walks* of arbitrarily small (negative) value

    This happens *iff* we reach a negative cycle that we can repeat indefinitely, always improving the total “cost” of the walk.



The Trouble with Negative Cycles

- The complications in the definition all stem from **negative-weight edges**

  $\delta_G(s,v) \;=\; \boxed{\inf\left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\right)}$

- In general, $\delta_G(s,v)$ can be
  - $+\infty$ if there is no s-v-walk at all, or (“no-path case” easy to detect and handle)
  - $-\infty$ if there are s-v-*walks* of arbitrarily small (negative) value

    This happens *iff* we reach a negative cycle that we can repeat indefinitely, always improving the total “cost” of the walk.

$\rightsquigarrow$ **Lemma (Shortest Paths):** If w is a shortest s-v-**walk** in $G = (V, E, c)$, there is an s-v-**path** p with $c(p) = c(w)$.

The Trouble with Negative Cycles

- The complications in the definition all stem from **negative-weight edges**

$$\delta_G(s,v) \;=\; \boxed{\inf\left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\right)}$$

- In general, $\delta_G(s,v)$ can be
  - $+\infty$ if there is no s-v-walk at all, or (“no-path case” easy to detect and handle)
  - $-\infty$ if there are s-v-*walks* of arbitrarily small (negative) value

    This happens *iff* we reach a negative cycle that we can repeat indefinitely, always improving the total “cost” of the walk.

$\rightsquigarrow$ **Lemma (Shortest Paths):** If w is a shortest s-v-**walk** in $G = (V, E, c)$, there is an s-v-**path** p with $c(p) = c(w)$.

Proof: Suppose w contains a cycle C.

- If $c(C) < 0$, w is not shortest as we can repeat C and reduce cost ↯
- If $c(C) > 0$, w is not shortest as we can remove C and reduce cost ↯
- If $c(C) = 0$ for all cycles in w, we can remove them from w to obtain a path p and $c(p) = c(w)$.

The Trouble with Negative Cycles

- The complications in the definition all stem from **negative-weight edges**

$$\delta_G(s,v) \;=\; \boxed{\inf\left(\{+\infty\} \cup \{c(w) : w \text{ an } s\text{-}v\text{-walk in } G\}\right)}$$

- In general, $\delta_G(s,v)$ can be
  - $+\infty$ if there is no s-v-walk at all, or (“no-path case” easy to detect and handle)
  - $-\infty$ if there are s-v-*walks* of arbitrarily small (negative) value

    This happens *iff* we reach a negative cycle that we can repeat indefinitely,
    always improving the total “cost” of the walk.

⇝ **Lemma (Shortest Paths):** If w is a shortest s-v-**walk** in $G = (V, E, c)$,
there is an s-v-**path** p with $c(p) = c(w)$.

Proof: Suppose w contains a cycle C.

- If $c(C) < 0$, w is not shortest as we can repeat C and reduce cost ↯
- If $c(C) > 0$, w is not shortest as we can remove C and reduce cost ↯
- If $c(C) = 0$ for all cycles in w, we can remove them from w to obtain a path p and $c(p) = c(w)$.

⇝ In the absense of negative cycles, all shortest walks are **shortest paths** (of at most $n-1$ edges).

Variants of Shortest Path Problems

Important special cases

1. **Positive SSSPP**
   - $c : E \to \mathbb{R}_{>0}$
   - most relevant case for many applications $\rightsquigarrow$ focus of this section
2. **Unweighted SSSPP**
   - $c(e) = 1$ for $e \in E$ $\rightsquigarrow$ $c(w) =$ #edges for every walk w

   $\rightsquigarrow$ solved by BFS in linear time
3. **Acyclic SSSPP**
   - G is a DAG
   - can be solved in linear time based on topological sort (for *arbitrary* c)

Variants of Shortest Path Problems

Important special cases

1. **Positive SSSPP**
   - $c : E \to \mathbb{R}_{>0}$
   - most relevant case for many applications $\rightsquigarrow$ focus of this section
2. **Unweighted SSSPP**
   - $c(e) = 1$ for $e \in E$ $\rightsquigarrow$ $c(w) =$ #edges for every walk w

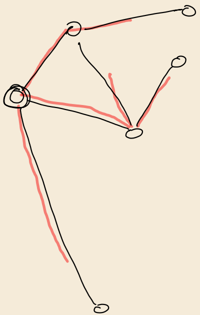
   $\rightsquigarrow$ solved by BFS in linear time
3. **Acyclic SSSPP**
   - G is a DAG
   - can be solved in linear time based on topological sort (for *arbitrary* c)

- For the rest of this section, we will assume $c(e) > 0$.
- But: The general case of cyclic graphs with negative edge weights is also relevant
  - We will come back to this case in Unit 12!

Dijkstra's Algorithm

- **Intuition:** Imagine sending out many little pioneers, walking at unit speed from s across all edges in G. The first pioneer to reach a vertex v "claims" v and proclaims the current time (= distance).

  Dijkstra's Algorithm is a event-driven simulation of this process!



Dijkstra's Algorithm

- **Intuition:** Imagine sending out many little pioneers, walking at unit speed from s across all edges in G. The first pioneer to reach a vertex v "claims" v and proclaims the current time (= distance).

  Dijkstra's Algorithm is a event-driven simulation of this process!

  - Event: Some pioneer reaches a new vertex.
    Can set a "timer" for that as soon as they start walking over an edge.
  - Maintain priority queue of events, sorted by time.
    - Discard events for vertices that have been claimed already.
    - Avoid generating events when already clear that they will be discarded.
  - Note: With $c(e) = 1$, this simulates BFS!

Dijkstra's Algorithm

- **Intuition:** Imagine sending out many little pioneers, walking at unit speed from s across all edges in G.
  The first pioneer to reach a vertex v "claims" v and proclaims the current time (= distance).

  Dijkstra's Algorithm is a event-driven simulation of this process!

  - Event: Some pioneer reaches a new vertex.
    Can set a "timer" for that as soon as they start walking over an edge.
  - Maintain priority queue of events, sorted by time.
    - Discard events for vertices that have been claimed already.
    - Avoid generating events when already clear that they will be discarded.
  - Note: With $c(e) = 1$, this simulates BFS!

- **Implementation:** Store unclaimed vertices in IndexMinPQ
  Priority = earliest time known so far when this vertex will be claimed

  - To claim w at time t, must have claimed some v at time $t - c(vw)$

  $\rightsquigarrow$ whenever we claim a vertex v, update successors' claim times (via `decreaseKey`)

Dijkstra's Algorithm

- **Intuition:** Imagine sending out many little pioneers, walking at unit speed from s across all edges in G. The first pioneer to reach a vertex v "claims" v and proclaims the current time (= distance).

  Dijkstra's Algorithm is a event-driven simulation of this process!

  - Event: Some pioneer reaches a new vertex.
    Can set a "timer" for that as soon as they start walking over an edge.
  - Maintain priority queue of events, sorted by time.
    - Discard events for vertices that have been claimed already.
    - Avoid generating events when already clear that they will be discarded.
  - Note: With $c(e) = 1$, this simulates BFS!

- **Implementation:** Store unclaimed vertices in IndexMinPQ
  Priority = earliest time known so far when this vertex will be claimed

  - To claim w at time t, must have claimed some v at time $t - c(vw)$

  $\rightsquigarrow$ whenever we claim a vertex v, update successors' claim times (via `decreaseKey`)

  $\rightsquigarrow$ overall process is a graph traversal! claimed = ***done***

Dijkstra's Algorithm – Code & Correctness

```
procedure dijkstra(G):
    // Assume G = (V, E, c) is simple (di)graph, c : E → ℝ_{>0}
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := +∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0);  dist[0] := 0
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return (dist, father)

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c > 0
        if ¬inS[w]
            if dist[v] + c < dist[w]
                // s ⇝ v → w new currently cheapest path to w
                father[w] := v;  dist[w] := dist[v] + c
                if Q.contains(w) then Q.decreaseKey(w, c)
                else Q.insert(w, c) end if // w active
            end if
        end if
    end for
    inS[v] := true // v done
```

- Same as primMST except *dist* computation *distance from* s, *not distance from* S

Dijkstra's Algorithm – Code & Correctness

```
procedure dijkstra(G):
    // Assume G = (V, E, c) is simple (di)graph, c : E → ℝ>0
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := +∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0);  dist[s] := 0
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return (dist, father)

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c > 0
        if ¬inS[w]
            if dist[v] + c < dist[w]
                // s ⇝ v → w new currently cheapest path to w
                father[w] := v;  dist[w] := dist[v] + c
                if Q.contains(w) then Q.decreaseKey(w, c)
                else Q.insert(w, c) end if // w active
            end if
        end if
    end for
    inS[v] := true // v done
```

- Same as primMST except *dist* computation *distance from* s, *not distance from* S

⇝ **Same running time:**

- $n \times$ `insert`, $(n-1) \times$ `delMin`, up to $m \times$ `decreaseKey`

⇝ with binary heaps $O(m \log n)$
with Fibonacci heaps $O(m + n \log n)$

Dijkstra's Algorithm – Code & Correctness

```
procedure dijkstra(G):
    // Assume G = (V, E, c) is simple (di)graph, c : E → ℝ>0
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := +∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0);  dist[s] := 0
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return (dist, father)

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c > 0
        if ¬inS[w]
            if dist[v] + c < dist[w]
                // s ⇝ v → w new currently cheapest path to w
                father[w] := v;  dist[w] := dist[v] + c          dist[w]
                if Q.contains(w) then Q.decreaseKey(w, ~~c~~)
                else Q.insert(w, ~~c~~) end if // w active
            end if              dist[w]
        end if
    end for
    inS[v] := true // v done
```

- Same as primMST except *dist* computation *distance from s, not distance from S*

⇝ **Same running time:**

  - $n \times$ `insert`, $(n-1) \times$ `delMin`, up to $m \times$ `decreaseKey`

  ⇝ with binary heaps $O(m \log n)$
  with Fibonacci heaps $O(m + n \log n)$

- **Correctness:**

  1. current "time" = $dist[v]$ in visit(v) calls strictly increasing over iterations

Dijkstra's Algorithm – Code & Correctness

```
procedure dijkstra(G):
    // Assume G = (V, E, c) is simple (di)graph, c : E → ℝ_{>0}
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := +∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0);  dist[s] := 0
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return (dist, father)

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c > 0
        if ¬inS[w]
            if dist[v] + c < dist[w]
                // s ⇝ v → w new currently cheapest path to w
                father[w] := v;  dist[w] := dist[v] + c
                if Q.contains(w) then Q.decreaseKey(w, c)
                else Q.insert(w, c) end if // w active
            end if
        end if
    end for
    inS[v] := true // v done
```

- Same as primMST except *dist* computation
  distance from s, *not distance from* S

⇝ **Same running time:**

- $n \times$ `insert`, $(n-1) \times$ `delMin`,
  up to $m \times$ `decreaseKey`

⇝ with binary heaps $O(m \log n)$
  with Fibonacci heaps $O(m + n \log n)$

- **Correctness:**

1. current "time" = $dist[v]$ in visit(v) calls strictly increasing over iterations
2. Invariant: $dist[v]$ is cost of *some* s-v-path
   or $dist[v] = +\infty$

Dijkstra's Algorithm – Code & Correctness

```
procedure dijkstra(G):
    // Assume G = (V, E, c) is simple (di)graph, c : E → ℝ_{>0}
    father[0..n) := NONE;  inS[0..n) := false;  dist[0..n) := +∞
    Q := new IndexMinPQ(n)
    Q.insert(0,0);  dist[s] := 0
    while ¬Q.isEmpty()
        visit(Q.delMin())
    return (dist, father)

procedure visit(v):
    for (w, c) ∈ G.adj[v] // edge vw with cost c > 0
        if ¬inS[w]
            if dist[v] + c < dist[w]
                // s ⇝ v → w new currently cheapest path to w
                father[w] := v;  dist[w] := dist[v] + c
                if Q.contains(w) then Q.decreaseKey(w, c)
                else Q.insert(w, c) end if // w active
            end if
        end if
    end for
    inS[v] := true // v done
```

- Same as primMST except *dist* computation *distance from* s, *not distance from* S

⇝ **Same running time:**

- $n \times$ `insert`, $(n-1) \times$ `delMin`, up to $m \times$ `decreaseKey`

⇝ with binary heaps $O(m \log n)$
with Fibonacci heaps $O(m + n \log n)$

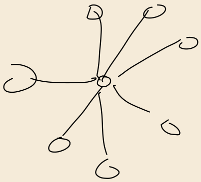
- **Correctness:**

1. current "time" = $dist[v]$ in visit(v) calls strictly increasing over iterations
2. Invariant: $dist[v]$ is cost of *some* s-v-path or $dist[v] = +\infty$
3. $dist[u] = \delta_G(s, u)$ for all $u \in$ ***done***

Shortest Paths Discussion

- 👍 Simple and efficient solution if edge weights are positive
- 👍 Dijkstra's Algorithm (with Fibonacci heaps) is worst-case optimal $\Theta(m + n \log n)$
  - ▶ (for sorting vertices by distance from s in a comparison-addition model)
  - ▶ another fine example of a greedy algorithm!

can reduce sorting
to SSSPP



Shortest Paths Discussion

- Simple and efficient solution if edge weights are positive
- Dijkstra's Algorithm (with Fibonacciy heaps) is worst-case optimal
  - (for sorting vertices by distance from s in a comparison-addition model)
  - another fine example of a greedy algorithm!

- improvements often possible for s-t shortest paths (although worst case remains same)
  - in SSSPP Dijkstra, can stop once t is ***done***
  - bidirectional Dijkstra (alternatingly work from both ends until we "meet")
  - A^*/goal-directed search (use cheap lower bound for $\delta_G(v,t)$ in vertex selection)
- we will revisit the general SSSPP (with negative weights)