



School of Computer Science and Engineering
COMP3121/9101 Algorithm Design and Analysis

Module 2: Greedy Algorithms

Raveen de Silva
K17-202

Course Admins:
cs3121@unsw.edu.au
(Liam, Leith, Hugo, Stephen, James, Brendan)

2026 T3

Overview

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Example problem: Frozen Yoghurt I

Problem: Frozen Yoghurt I

There are n flavours of frozen yoghurt at your local shop. The i th flavour is dispensed from a machine of capacity c_i litres and contributes d_i/c_i deliciousness per litre.

You have a giant tub of capacity C litres. You want to fill the tub so as to maximise the total deliciousness.

Equivalent to a well-studied problem:

Problem: Fractional Knapsack

Instance: A list of n items described by their weights w_i and values v_i , and a weight limit W of your knapsack.

You can take any fraction between 0 and 1 of each item.

Task: Select a non-negative quantity of each item, with total weight not exceeding W and maximum total value.

How do we choose which items to take?

- ▶ Items with low weight are better.
- ▶ Items with high value are better.

Fractional Knapsack

Solution

Fill the entire knapsack with the item of highest value per unit weight!

If you run out of that item, use the second-best item by this ranking, and so on.

In other words, select the item with the largest ratio $\frac{v_i}{w_i}$ and pack as much of that item as you can, before trying the pack the item with the next largest ratio.

Example problem: Frozen Yoghurt II

Problem: Frozen Yoghurt II

Sadly, the frozen yoghurt shop has now closed down, so you must go to the supermarket instead. There, you can find n flavours of frozen yoghurt in tubs. The i th flavour comes in a tub of capacity c_i litres and contributes d_i deliciousness.

You again want to buy C litres, with maximum total deliciousness.

Equivalent to a well-studied problem:

Problem: 0-1 Knapsack

Instance: A list of n **discrete items** described by their weights w_i and values v_i , and a weight limit W of your knapsack.

Task: Find a subset S of the items with total weight not exceeding W and maximum total value.

- ▶ As before, we want **large value** and **low weight** items.
- ▶ But this time we can't take fractions of an item, and we can't take multiple copies of an item. How does that change things?

0-1 Knapsack

- ▶ Can we always choose the item of highest value per unit weight?
- ▶ No. Assume there are just three items with weights and values

A : 1 kg, \$6,

B : 2 kg, \$10,

C : 3 kg, \$12,

and a knapsack of capacity $W = 5$ kg.

The greedy strategy would choose items A and B , while the optimal solution is to take items B and C !

- ▶ So when do greedy algorithms work?
- ▶ Unfortunately there is no easy rule ...

P versus NP (spoiler for Intractability module)

For those interested, it relates to the following problem:

Open problem (\$1,000,000 prize if you solve it!)

Is $P = NP$?

- ▶ Fractional knapsack is a problem in P.
- ▶ 0-1 knapsack is an NP-hard problem.
- ▶ A greedy algorithm (or any other polynomial-time algorithm) for the 0-1 knapsack problem would imply $P = NP$, which is widely believed to be false, and this would solve the most important open problem in computer science.

Question

What is a greedy algorithm?

Answer

A greedy algorithm is an algorithm that solves a problem by dividing it into stages, and rather than exhaustively searching all the ways to get from one stage to the next, instead only considers the choice that appears best.

This obviously reduces the search space, but it is not always clear whether the locally optimal choice leads to the globally optimal outcome.

Question

Are greedy algorithms always correct?

Answer

No!

Suppose you are searching for the highest point in a mountain range. If you always climb upwards from the current point in the steepest possible direction, you will find a peak, but not necessarily the highest point overall.

Greedy Algorithms

Question

Is there a framework to decide whether a problem can be solved using a greedy algorithm?

Answer

Yes: matroids (for a narrow definition of greedy) and matroid embeddings (for a more inclusive definition of greedy). It requires finding/defining the right combinatorial structure, which is not necessarily easier than formulating a greedy algorithm directly.

Matroids are covered in COMP3821/9801. We will instead prove the correctness of greedy algorithms on a problem-by-problem basis. With experience, you will develop an intuition for whether greedy algorithms are useful for a particular problem.

Greedy Algorithms

- ▶ Many greedy algorithms are relatively simple to describe.
- ▶ The trade-off is that they by definition don't consider all of the possibilities.
- ▶ This leaves some work to do in analysing correctness.
- ▶ Why did you not need to consider those other possibilities? Equivalently, why were the possibilities you considered no worse than the others?

Question

How do we prove that a greedy algorithm is correct?

We need to show that it produces solutions that are **valid** (usually easy) and **optimal**.

Answer

There are two main methods of proof:

1. **Greedy stays ahead**: prove that at every stage, no other sequence of choices could do better than our proposed algorithm.
2. **Exchange argument**: consider any alternative solution, and gradually transform it to the solution found by our proposed algorithm without making it any worse.

Note: instead of starting from any alternative solution, we can instead start from any optimal solution.

These are reminiscent of induction and contradiction respectively.

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Activity Selection

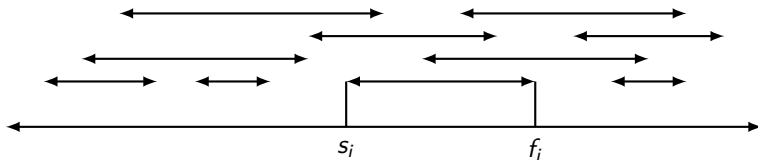
Problem: Activity Selection

Instance: A list of n activities, with starting times s_i and finishing times f_i .

Task: Find a largest number of pairwise compatible activities. Two activities are **compatible** if their intervals do not overlap.

Possible ideas:

- ▶ Choose shorter activities first?
- ▶ Choose activities which have fewest conflicts?
- ▶ Choose activities with earlier end times first?



Activity Selection

Attempt 1

Always choose the shortest activity which does not overlap with any previously chosen activities, then remove the activities that overlap with the current choice and repeat.

Does this produce solutions that are valid (Yes) and optimal (No)?



In this example, our proposed algorithm chooses the shortest activity (blue), then has to discard both the red activities, which is worse than picking both of the longer activities.

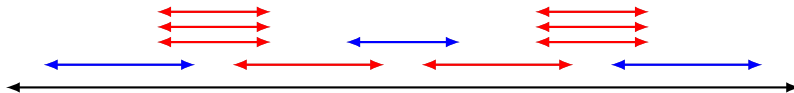
Possible ideas:

- ▶ ~~Choose shorter activities first?~~
- ▶ Choose activities which have fewest conflicts?
- ▶ Choose activities with earlier end times first?

Activity Selection

Attempt 2

Always choose an activity which conflicts with the fewest possible number of the remaining activities? Remove the activities that overlap with the current choice and repeat.



As appealing this idea is, the above figure shows this again does not work!

Possible ideas:

- ▶ ~~Choose shorter activities first?~~
- ▶ ~~Choose activities which have fewest conflicts?~~
- ▶ Choose activities with earlier end times first?

Activity Selection

Attempt 3 (Solution)

Among those activities which do not conflict with the previously chosen activities, always choose the activity with the earliest end time (breaking ties arbitrarily). Remove the activities that overlap with the current choice and repeat.

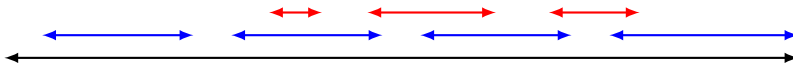


Activity Selection

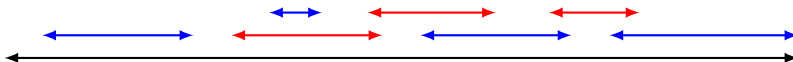
To prove the correctness of our algorithm, we will use an **exchange argument**. We will show that any valid alternative selection can be transformed into the selection obtained by our greedy algorithm with at least as many activities and without introducing conflicts.

- ▶ Start with an arbitrary valid alternative selection.
(The same proof goes through if we start with an optimal valid alternative selection.)
- ▶ Find the first place where the chosen activity violates the greedy choice.
- ▶ Can we replace that activity with the greedy choice?

Alternative selection:



Modified alternative:



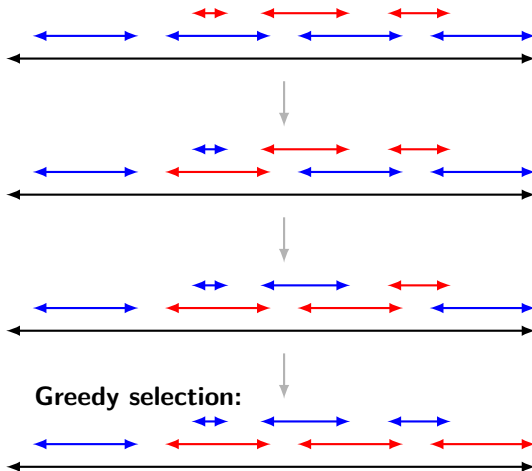
Activity Selection: proof outline

- ▶ Does the new selection have any conflicts?
 - No!
- ▶ Does the new selection have the same number of activities?
 - Yes!
- ▶ So where our alternative differs from the greedy selection, a modified alternative which makes the same choice as the greedy is both valid and just as good.

Activity Selection: proof outline

- ▶ We replace this activity and repeat for the next activity that differs.
- ▶ Continuing in this manner, we can eventually “morph” any alternative selection until it has no disagreements with the greedy selection.
- ▶ At the end of this process, we will have a selection that:
 - is valid,
 - has the same number of activities as the alternative selection we started with, and
 - has no activities other than those in the greedy selection.
- ▶ So the greedy algorithm produces a valid selection that contains at least as many activities as **any** alternative selection does, i.e., this greedy algorithm is optimal.

Alternative selection:



Greedy selection:

Activity Selection: formal correctness proof

- ▶ Suppose the greedy algorithm selects activities $G = \{g_1, \dots, g_r\}$, and consider some alternative valid selection $A = \{a_1, \dots, a_s\}$, each in ascending order of finishing time.
 - Our aim is to prove that no other valid selection can have more activities than the greedy solution, i.e., $s \leq r$.
- ▶ Suppose that the smallest index where G and A disagree is k . So they both select the same first $k - 1$ activities. In other words, assume

$$g_1 = a_1, g_2 = a_2, \dots, g_{k-1} = a_{k-1}, \text{ but } g_k \neq a_k.$$

- ▶ Then we will compare

$$A = \{g_1, \dots, g_{k-1}, a_k, a_{k+1}, \dots, a_s\}$$

with

$$A' = \{g_1, \dots, g_{k-1}, g_k, a_{k+1}, \dots, a_s\}.$$

Activity Selection: correctness

$$A' = \{g_1, \dots, g_{k-1}, g_k, a_{k+1}, \dots, a_s\}.$$

Claim

A' is valid.

Proof.

- ▶ There are no conflicts among $\{g_1, \dots, g_{k-1}, a_{k+1}, \dots, a_s\}$; these were all in the valid selection A .
- ▶ g_k doesn't conflict with $\{g_1, \dots, g_{k-1}\}$: the algorithm removes all activities conflicting with $g_1, \dots, g_{k-1}$ before choosing g_k .
- ▶ g_k finishes no later than a_k , which in turn finishes before any of $a_{k+1}, \dots, a_s$ start. So, g_k doesn't conflict with $\{a_{k+1}, \dots, a_s\}$.
- ▶ Therefore, A' has no conflicts, i.e., it is a valid selection.



Activity Selection: correctness

$$A' = \{g_1, \dots, g_{k-1}, g_k, a_{k+1}, \dots, a_s\}.$$

Claim

A' and A contain the same number of activities.

Proof.

- ▶ g_k isn't a duplicate of any of $\{g_1, \dots, g_{k-1}\}$ as G has no duplicates.
- ▶ g_k isn't a duplicate of any of $\{a_{k+1}, \dots, a_s\}$ as it finishes before they start.
- ▶ Therefore A' also contains s activities, i.e., $|A'| = |A|$.



Activity Selection: correctness

- ▶ We started with an alternative valid selection A , and the first time it disagreed with G we showed that the modified alternative A' without this disagreement is valid and has the same number of activities: $|A| = |A'|$ (the relevant information is that $|A'| \geq |A|$).
- ▶ We can now consider the first point of disagreement between G and A' , and repeat the argument.
- ▶ Continuing in this way, we resolve all disagreements between the greedy and the alternative selections until one of them runs out of activities.
- ▶ The greedy can't run out first: if a_{r+1} doesn't conflict with $g_1, \dots, g_r$, then the greedy selection wouldn't end with g_r (as it would contain another activity, e.g., a_{r+1} , afterwards).
- ▶ Therefore,

$$s = |A| \leq |A'| \leq \dots \leq |G| = r,$$

i.e., the alternative selection A has no more activities than our greedy selection G .

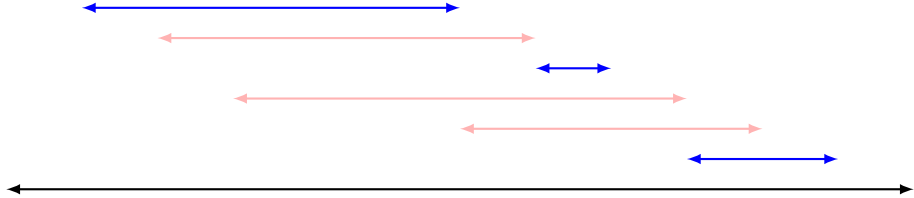
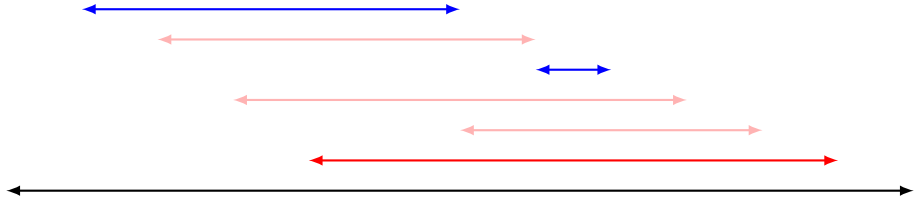
- ▶ Since A is an arbitrary alternative selection, we have that G contains at least as many activities as any other selection.
- ▶ Therefore, G is valid and optimal.

Activity Selection: time complexity

What is the time complexity of the algorithm?

- ▶ We represent activities by ordered pairs of their starting and their finishing times and sort them in increasing order of their finishing time (the second coordinate). Using mergesort, this takes $O(n \log n)$ time.
- ▶ We go through this sorted list in order. How do we tell whether an activity conflicts with the already chosen activities?
- ▶ Suppose we are up to activity i with interval $[s_i, f_i]$, and all earlier finishing activities have already been processed.
- ▶ If all previously chosen activities finished before s_i , activity i can be chosen without a conflict.
- ▶ Otherwise, there will be a clash, so we discard activity i .
- ▶ We would prefer not to go through all previously chosen activities each time.

Activity Selection: time complexity



Activity Selection: time complexity

- ▶ We need only keep track of the latest finishing time among chosen activities.
- ▶ Since we process activities in increasing order of finishing time, this is just the finishing time of the last activity that was chosen.
- ▶ Every activity is therefore either chosen (and the last finishing time updated) or discarded in constant time, so this part of the algorithm takes $O(n)$ time.
- ▶ Our algorithm has two steps:
 1. **Sorting** takes $O(n \log n)$ time.
 2. **Choosing** takes $O(n)$ time.
- ▶ Thus, the time complexity is

$$O(n \log n) + O(n) = O(n \log n).$$

Activity Selection

A related problem

Instance: A list of n activities with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration.

Task: Find a subset of compatible (i.e., non-overlapping) activities of **maximum total duration**.

Solution

Since all activities are of the same duration, this is equivalent to finding a selection with a largest number of non conflicting activities, i.e., the previous problem.

Activity Selection

Question

What happens if the activities are not all of the same duration?

Solution

The greedy strategy no longer works. Our earlier exchange argument relied on any interval being just as good as any other for the objective, and this isn't the case anymore.

We can't prune the search space as aggressively in this variant, so we'll need an "efficient brute force" ... dynamic programming

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Cell Towers

Problem

Instance: Along the long, straight road from Balladonia (in the West) to Caiguna (in the East), houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstra's cell tower is 5km.

Task: Design an algorithm for placing the smallest number of cell towers alongside the road, that is sufficient to cover all houses.



Cell Towers

- ▶ Let's attempt a greedy algorithm, processing the houses west to east.
- ▶ The first house must be covered by some tower, which we place 5km to the east of this house.
- ▶ This tower may cover some other houses, but eventually we will reach a house that is out of range of this tower. We then place a second tower 5km to the east of that house.
- ▶ Continue in this way until all houses are covered.

? We assume for now that there is no house within 5km of Caiguna, to avoid this edge case. Think about how you could handle the edge case in both the design of the algorithm and the correctness proof.

Cell Towers

- ▶ At each house, we need to decide whether to place a new tower. This can be done in constant time by referring to the most recently created tower, which can itself be updated in constant time if necessary.
- ▶ Therefore, this algorithm runs in $O(n)$ time if the houses are provided in order, and $O(n \log n)$ time otherwise.

Cell Towers: correctness proof outline

- ▶ We prove optimality using **greedy stays ahead**
- ▶ The key claim is:

Claim

For all $k \geq 1$, the k th tower in the greedy selection is at least as far east as the k th tower in any other valid selection.

- ▶ If this is true, then it is impossible to cover all the houses using fewer towers.

Claim

For all $k \geq 1$, the k th tower in the greedy selection is at least as far east as the k th tower in any other valid selection.

Proof.

Our proof uses induction on k .

Base case: $k = 1$

- ▶ Greedy places the first tower 5km east of the first house.
- ▶ Suppose an alternative selection A has the first tower further east. Then it does not cover this first house. So, A is not valid.
- ▶ So, the claim is true for $k = 1$.

To prove the claim for each $k \geq 2$, we make an **inductive hypothesis**:

- ▶ Suppose the claim is true for $k - 1$. So $a_{k-1} \leq g_{k-1}$.

Inductive step: Consider tower k .

- ▶ Greedy placed towers at g_{k-1} and g_k . So there is a house h with

$$g_{k-1} + 5 < h = g_k - 5.$$

- ▶ The alternative selection placed towers at a_{k-1} and a_k . For the sake of contradiction, assume that $a_k > g_k$. Then,

$$a_{k-1} + 5 \leq g_{k-1} + 5 < h = g_k - 5 < a_k - 5.$$

- ▶ So house h is not covered by tower a_{k-1} nor tower a_k . Thus, A is not a valid selection – a contradiction. We conclude that $a_k \leq g_k$. □

Cell Towers: correctness proof

We have proved the claim:

Claim

For all $k \geq 1$, the k th tower in the greedy selection is at least as far east as the k th tower in any other valid selection.

We now use this claim to prove that no other valid selection uses fewer towers than greedy.

Proof that greedy is optimal.

- ▶ Suppose greedy uses r towers, placed at $g_1, \dots, g_r$.
- ▶ There is a house h at $g_r - 5$, which isn't covered by the previous tower. Therefore, $g_{r-1} + 5 < h$.
- ▶ From the claim, we know that for any other placement of $r - 1$ towers $a_1, \dots, a_{r-1}$, we have $a_{r-1} \leq g_{r-1}$.
- ▶ Therefore, house h cannot be covered in the first $r - 1$ towers in **any** placement of towers.
- ▶ So at least r towers are needed. Therefore, greedy is optimal.



Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Tape Storage

Problem

Instance: A list of n files of lengths L_i which have to be stored on a tape.

Each file is **equally likely** to be needed.

To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

Task: Order the files on the tape so that the average (expected) retrieval time is minimised.



Artwork credit: Stephen Chuang

Tape Storage: solution

- ▶ Any ordering of the files is a valid solution.
- ▶ Smallest files first seems sensible. Can we prove that this is optimal?
- ▶ Suppose we store the files in order $L_1, L_2, \dots, L_n$
- ▶ The probability we want to read any one of the files is $\frac{1}{n}$.
- ▶ So the expected retrieval time is:

$$\begin{aligned} E &= \sum_{i=1}^n \frac{1}{n} (L_1 + \dots + L_i) \\ &= \frac{1}{n} \sum_{i=1}^n (L_1 + \dots + L_i) \\ &= \frac{1}{n} (L_1 + (L_1 + L_2) + (L_1 + L_2 + L_3) + \dots + (L_1 + \dots + L_n)) \\ &= \frac{1}{n} (nL_1 + (n-1)L_2 + (n-2)L_3 + \dots + 2L_{n-1} + L_n). \end{aligned}$$

h

Tape Storage: solution

The expected retrieval time is

$$E = \frac{1}{n} (nL_1 + (n-1)L_2 + (n-2)L_3 + \cdots + 2L_{n-1} + L_n).$$

The factor $\frac{1}{n}$ will be present regardless of our chosen order of files, so for convenience we will discuss the “scaled” retrieval time

$$nL_1 + (n-1)L_2 + (n-2)L_3 + \cdots + 2L_{n-1} + L_n.$$

Claim

The scaled retrieval time is minimised if $L_1 \leq L_2 \leq L_3 \leq \dots \leq L_n$.

If this claim is true, we'll solve the problem by simply storing the files in ascending order of length.

Tape Storage: proof of optimality

Claim

The scaled retrieval time

$$nL_1 + (n-1)L_2 + (n-2)L_3 + \dots + 2L_{n-1} + L_n,$$

is minimised if $L_1 \leq L_2 \leq \dots \leq L_n$.

Proof (by contradiction).

- ▶ Suppose the retrieval time is minimised but the files are **not** in ascending order by length. Then there are two consecutive files with $L_i > L_{i+1}$.
- ▶ Swapping them changes the scaled retrieval time from

$$\begin{aligned} T &= \dots + (n-i+1)L_i + (n-i)L_{i+1} + \dots \\ \text{to } T' &= \dots + (n-i+1)L_{i+1} + (n-i)L_i + \dots \end{aligned}$$

- ▶ This decreases the scaled retrieval time because $T - T' = L_i - L_{i+1} > 0$, which contradicts the assumption that the scaled retrieval time was minimised before the swap.



Tape Storage: proof of optimality

Notes:

- ▶ Any arrangement which isn't sorted by length can be improved upon, so the best arrangement is the one where files are ordered from shortest to longest.
- ▶ Indeed, resolving these “adjacent inversions” one-by-one would gradually return us to the sorted sequence, (our proof uses the same idea as bubble sort!)
- ▶ Note that this doesn't mean our algorithm should sort the files using bubble sort.
 - We refer to bubble sort only because it is easy to analyse adjacent inversions; resolving them has only **local** effects.
 - We should use a more efficient sorting algorithm to actually sort the files.

Algorithm

Sort the files by increasing order of length using mergesort.

Tape Storage II

But what if the files are not equally likely to be picked?

Problem

Instance: A list of n files of lengths L_i and probabilities to be needed p_i , $\sum_{i=1}^n p_i = 1$, which have to be stored on a tape.

To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

Task: Order the files on the tape so that the **expected** retrieval time is minimised.

Now we have two criteria we could order by: length L_i and probability p_i .

- ▶ Files with small length L_i should appear near the start
- ▶ Files with large probability p_i should appear near the start.

How can we sort by both of these properties at the same time?

Tape Storage II: solution

Solution

Store files in decreasing value of the ratio $\frac{p_i}{L_i}$.

- ▶ If the files are stored in order $L_1, L_2, \dots, L_n$, then the expected retrieval time is

$$E = L_1 p_1 + (L_1 + L_2) p_2 + (L_1 + L_2 + L_3) p_3 \\ + \dots + (L_1 + L_2 + L_3 + \dots + L_n) p_n$$

Claim

The expected retrieval time E is minimised if the files are ordered by decreasing p_i/L_i .

Proof (1/2).

- ▶ For the sake of contradiction, suppose E is minimised, but there are two adjacent files k and $k + 1$ such that $\frac{p_k}{L_k} < \frac{p_{k+1}}{L_{k+1}}$.
- ▶ What happens if we swap the order of these two files?

Proof (2/2).

- ▶ Before the swap, the expected retrieval time is

$$\begin{aligned} E = & L_1 p_1 + (L_1 + L_2) p_2 + \dots \\ & + (L_1 + L_2 + \dots + L_{k-1} + L_k) p_k \\ & + (L_1 + L_2 + \dots + L_{k-1} + L_k + L_{k+1}) p_{k+1} \\ & + \dots + (L_1 + L_2 + \dots + L_n) p_n \end{aligned}$$

- ▶ After the swap, the new expected retrieval time is

$$\begin{aligned} E' = & L_1 p_1 + (L_1 + L_2) p_2 + \dots \\ & + (L_1 + L_2 + \dots + L_{k-1} + L_{k+1}) p_{k+1} \\ & + (L_1 + L_2 + \dots + L_{k-1} + L_{k+1} + L_k) p_k \\ & + \dots + (L_1 + L_2 + \dots + L_n) p_n. \end{aligned}$$

- ▶ Now, $E - E' = L_k p_{k+1} - L_{k+1} p_k$.

- ▶ We have

$$p_k / L_k < p_{k+1} / L_{k+1} \Leftrightarrow L_{k+1} p_k < L_k p_{k+1}.$$

- ▶ Thus, $E - E' =$

$$L_k p_{k+1} - L_{k+1} p_k > 0.$$

- ▶ Therefore, $E > E'$. This is a contradiction, since E is not the smallest possible retrieval time.

- ▶ Therefore, there is no **inversion** (file $k+1$ with a larger ratio p_{k+1}/L_{k+1} occurs after file k with a smaller ratio p_k/L_k).

- ▶ So the optimal sequence is the one with no inversions, i.e., in descending order of the ratio p_i/L_i .



Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Minimising Job Lateness

Problem

Instance: A list of n jobs, with lengths len_i and deadlines due_i .

All jobs have to be completed, but only one job can be performed at any time.

If a job i is completed at a finishing time fin_i then we say that it has incurred lateness $late_i = fin_i - due_i$.

Task: Schedule all the jobs to minimise the largest lateness.

Any schedule that leaves gaps between two consecutive jobs can be turned into a schedule that has no gaps without increasing any lateness.

All valid solutions can therefore be described by an ordering of the jobs.

We have two criteria we can order by: length and deadline. Which is more important to find the optimal solution?

- ▶ Perform quicker tasks first? (small len_i)
- ▶ Prioritise tasks with earlier deadlines? (small due_i)
- ▶ Maybe we need to consider both factors?

Minimising Job Lateness

Actually, the lengths have no effect on the optimal solution.

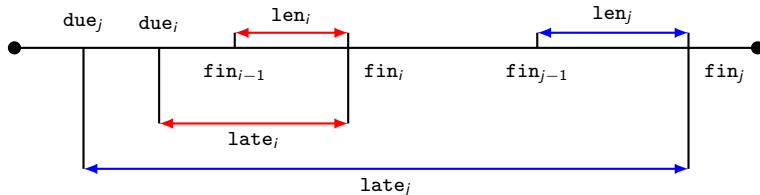
Solution

Schedule jobs in ascending order of deadlines, due_i .

Proof of optimality (1/3).

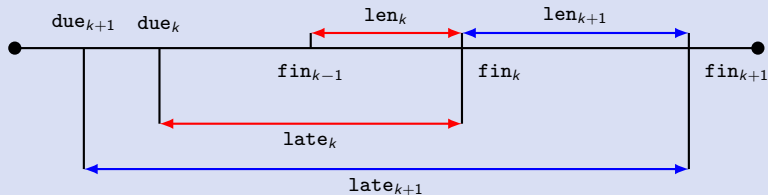
We use an exchange argument to show that an ordering by ascending deadlines is optimal.

- ▶ Consider any optimal ordering of jobs $A = [1, 2, 3, 4, \dots, n]$ that is not sorted.
- ▶ We will show that we can turn A into an ordering sorted by ascending deadlines without increasing the maximum lateness.
- ▶ We say jobs i and j form an inversion if job i is scheduled before job j but is due after it.

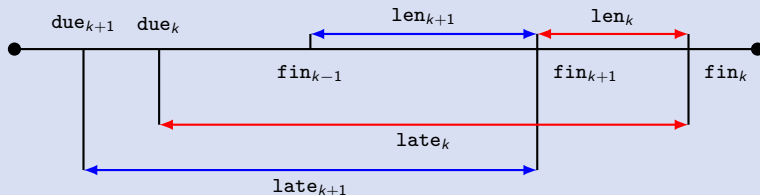


Proof of optimality (2/3).

- ▶ Since A is not sorted, it has an inversion. It even has an adjacent inversion.
- ▶ There are two consecutive jobs $k, k + 1$ in ordering A so that k is scheduled immediately before $k + 1$, but $\text{due}_{k+1} < \text{due}_k$.



- ▶ What happens to the lateness of k and $k + 1$ when we swap their order?



Proof of optimality (3/3).

- ▶ Before the swap, $\text{late}_{k+1} > \text{late}_k$. The swap decreases late_{k+1} and increases late_k . Importantly, the value of late_k after the swap is not greater than the value of late_{k+1} before the swap, as the finishing times are the same but $\text{due}_{k+1} < \text{due}_k$.
- ▶ The swap does not change the lateness of any other jobs.
- ▶ Therefore, the largest lateness overall does not increase by swapping adjacent inversions.
- ▶ Starting from the optimal ordering A and repeatedly swapping adjacent inversions produces an optimal ordering B that is sorted by ascending deadlines and whose lateness is no greater than the lateness of A .
- ▶ To conclude that the solution C returned by the greedy algorithm is optimal (C could differ from B if there are ties), it suffices to observe that any two sorted orderings have the same maximum lateness.



Note: it would have been much harder to directly analyse **non-adjacent** inversions. Swapping non-adjacent jobs would also affect the finishing time and hence lateness of every job in between them.

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Array Merging

- ▶ Assume you are given n sorted arrays of different sizes.
- ▶ You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left.

Exercise

Design an algorithm which achieves this task and moves array elements as few times as possible.

Give an informal justification why your algorithm is optimal.

This problem is somewhat related to the next problem, which is arguably among the most important applications of the greedy method!

Character Encoding

Problem: Unambiguous code

You are given a set of symbols, for example the English alphabet plus some punctuation marks and a blank space (to be used between words), giving the 31 symbols:

abcdefghijklmnopqrstuvwxyz ,.?!

You want to encode these symbols using binary strings, so that sequences of such symbols can be decoded in an unambiguous way.

- ▶ “Unambiguous” means there is only one way to decode the binary string.
original message $\rightarrow$ binary code $\rightarrow$ decoded message
- ▶ How long do these binary strings need to be?
- ▶ We have 31 symbols and $2^5 = 32 > 31$, so 5 bits per symbol is sufficient

Fixed-width Encoding

A possible solution

Assign a different 5 bit sequence to each symbol in

abcdefghijklmnopqrstuvwxyz ,.?!

- ▶ This works! Every symbol can be encoded, and any message we encode can be decoded unambiguously.
- ▶ To decode you partition the bitstream into groups of 5 bits and use a lookup table.
- ▶ But this code is inefficient. Common symbols and rare symbols are all given binary encodings of the same length.
- ▶ It would be more efficient to encode common symbols with short strings.
- ▶ Is there a way to construct an efficient and unambiguous code?

Example: variable width encoding

- ▶ Let's look at a smaller alphabet with just 6 symbols:

acekts

- ▶ Since $2^3 = 8 > 6$, we could assign each symbol a 3-bit binary string.
- ▶ But the letters *e* and *t* are very common in English, so let's give them shorter strings:

$e = 00, t = 11$

- ▶ And all the other letters get strings of length 3:

$a = 010, c = 101, k = 111, s = 100$

- ▶ Is this binary code unambiguous? No. Let's see why

Example: variable width encoding

Code: $a = 010, c = 101, e = 00, k = 111, s = 100, t = 11$

Original message: *cake*

original message $\rightarrow$ binary code $\rightarrow$ decoded message

Encoding step.

cake $\rightarrow (101)ake \rightarrow (101)(010)ke \rightarrow (101)(010)(111)e \rightarrow (101)(010)(111)(00)$

Binary code. 10101011100

Decoding step.

Two possible decodings:

- ▶ 101 010 11 100 $\rightarrow$ *cats*
- ▶ 101 010 111 00 $\rightarrow$ *cake*

Our code was ambiguous because the encoding of $t = 11$ is a **prefix** of $k = 111$.

Prefix Codes

- ▶ A code in which no coded symbol is a prefix of another is called a **prefix code**.
- ▶ Prefix codes are unambiguous.

Problem: efficient, unambiguous code

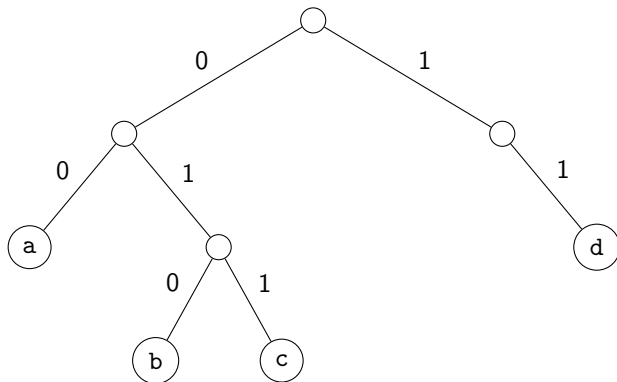
You are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words), giving the 31 symbols:

abcdefghijklmnopqrstuvwxyz ,.?!

Each symbol has a frequency (probability of occurring). You want to encode these symbols using binary strings, so that sequences of such symbols can be decoded in an unambiguous way, and so that the expected length of the encoded text is minimised.

Prefix Codes

- ▶ Any encoding into binary can be visualised using a binary tree, where left branches represent '0' and right branches '1'.
- ▶ In a prefix code, no ancestor of a codeword is itself a codeword, so all codewords must be located at leaves.



Optimal solution

Huffman code!

- ▶ Huffman's method uses a greedy algorithm to construct an optimal prefix code.
- ▶ This makes sure that the average number of bits in an encoding of an “average” text is as small as possible.
- ▶ We'll next go through an example of how to apply Huffman's code. For the details of the algorithm, and the proof of optimality, see the textbook.
- ▶ MATH3411 Information, Codes and Ciphers covers this and much more!

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

The Huffman Code

Main idea.

- ▶ Make a priority queue of symbols, each weighted by its frequency.
- ▶ Repeatedly combine the two lowest frequency symbols.
 - Remove the two symbols of lowest frequency, say x and y with frequencies f_x and f_y .
 - Make a new symbol xy with frequency $f_x + f_y$, and insert it to the priority queue.
- ▶ As we do this, we maintain a forest, initially consisting of n isolated vertices.
 - When symbols x and y are combined, we add a new vertex representing xy , whose left and right children are x and y .
 - This records that the codeword for x is to be found by appending a '0' to the codeword for xy , and likewise for y by appending a '1'.

The Huffman Code: example

symbol	a	b	c	d	e
frequency	0.32	0.25	0.20	0.18	0.05

a

b

c

d

e

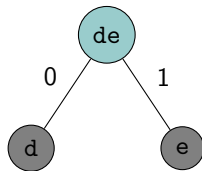
The Huffman Code: example

symbol	a	b	c	d	e	de
frequency	0.32	0.25	0.20	0.18	0.05	0.23

a

b

c



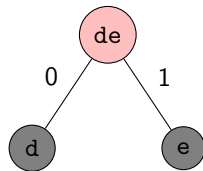
The Huffman Code: example

symbol	a	b	c	de
frequency	0.32	0.25	0.20	0.23

a

b

c

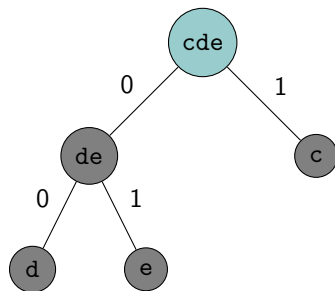


The Huffman Code: example

symbol	a	b	c	de	cde
frequency	0.32	0.25	0.20	0.23	0.43

a

b

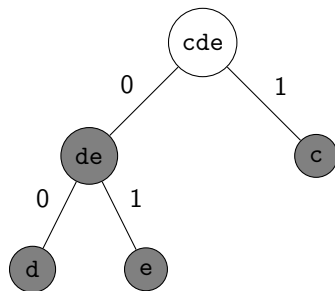


The Huffman Code: example

symbol	a	b	cde
frequency	0.32	0.25	0.43

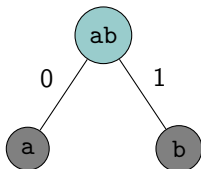
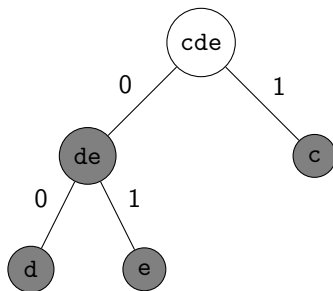
a

b



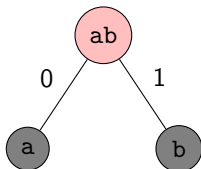
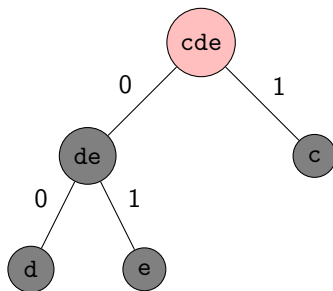
The Huffman Code: example

symbol	a	b	ab	cde
frequency	0.32	0.25	0.57	0.43



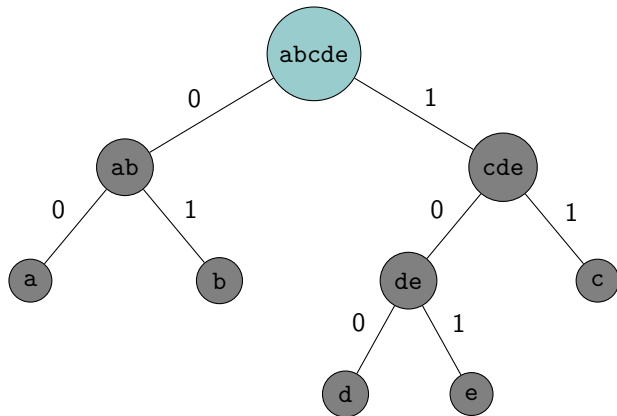
The Huffman Code: example

symbol	ab	cde
frequency	0.57	0.43



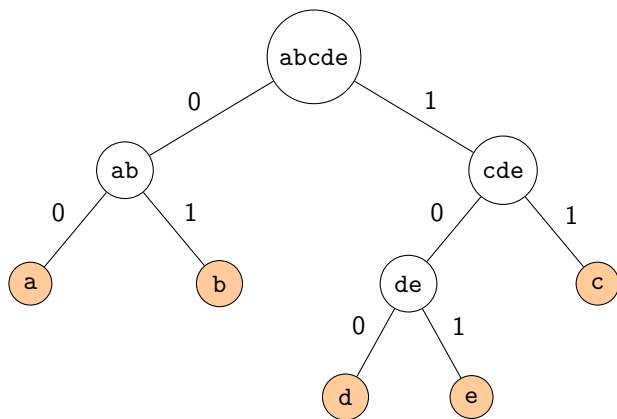
The Huffman Code: example

symbol	cde	ab	abcde
frequency	0.43	0.57	1.00



The Huffman Code: example

symbol	a	b	c	d	e
codeword	00	01	11	100	101



The Huffman Code: example

symbol i	a	b	c	d	e
codeword C_i	00	01	11	100	101
frequency f_i	0.32	0.25	0.20	0.18	0.05

The average codeword length is

$$\begin{aligned} L &= \sum_{i \in \{a,b,c,d,e\}} \text{len}(C_i) \times f_i \\ &= [\text{len}(C_a) \times f_a] + [\text{len}(C_b) \times f_b] + [\text{len}(C_c) \times f_c] \\ &\quad + [\text{len}(C_d) \times f_d] + [\text{len}(C_e) \times f_e] \\ &= 2 \times 0.32 + 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.18 + 3 \times 0.05 \\ &= 2.23 \end{aligned}$$

The Huffman Code: correctness outline

Lemma

Given the frequency f_i of each symbol i , the Huffman code produces an optimal prefix code.

1. First, prove that an optimal prefix code must arise from a **full** binary tree; one in which every node has exactly zero or two children.
 - Hopefully obvious; a node with only one child could be replaced by that child.
2. Then, prove that in some optimal tree, the two symbols of lowest frequency are siblings at the greatest depth.
 - If these two symbols aren't siblings at the greatest depth, then swap them with whatever symbols are.
 - Either nothing changes depth, or
 - less frequent symbols increase in depth and more frequent symbols decrease in depth by the same amount, so it is a net improvement.
3. Finally, prove (by induction on the number of symbols) that the Huffman code is optimal.
 - Perform the combine operation on some optimal tree and the Huffman tree to get trees on $n - 1$ vertices, and use the induction hypothesis.

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

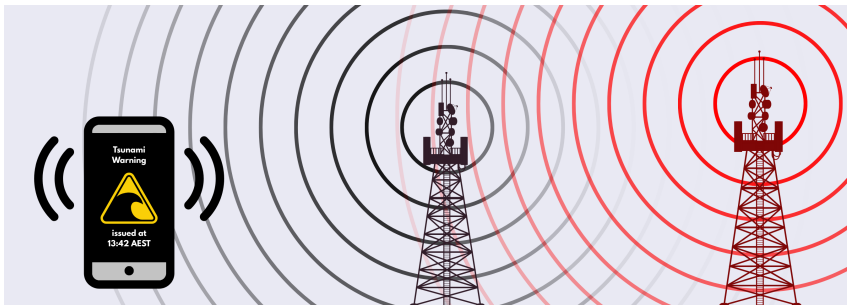
Problem: Tsunami Warning

Instance: There are n radio towers for broadcasting tsunami warnings. A tower i has coordinates (x_i, y_i) and a range r_i .

You need to equip some of these towers with seismic sensors.

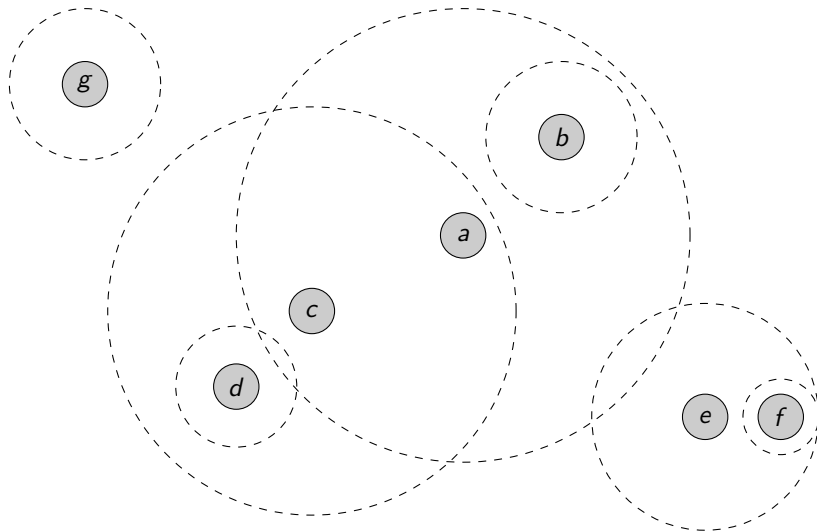
When there is a tsunami risk, all towers with sensors activate and each activated tower also activates all towers within its range, and so on.

Task: Design an algorithm that finds the smallest number of towers you must equip with seismic sensors so that all towers become activated.

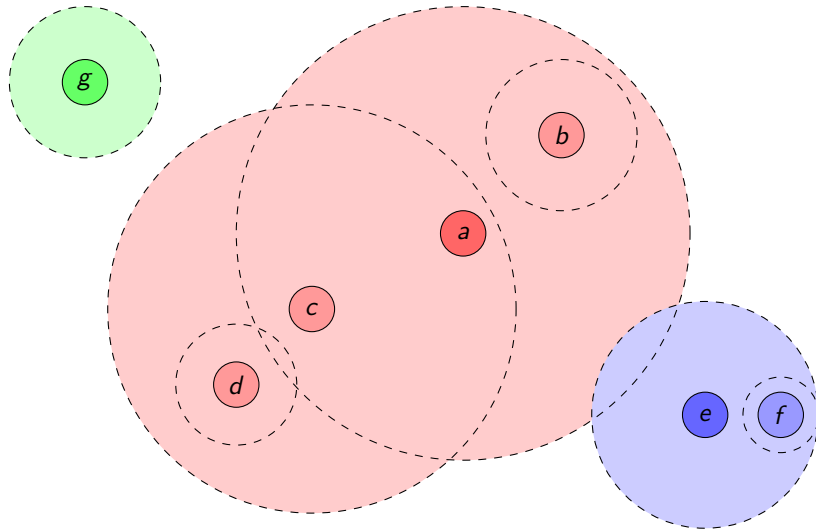


Artwork credit: Stephen Chuang

Tsunami Warning



Tsunami Warning



How to construct a greedy algorithm for placing sensors?

What inputs were we given?

- ▶ Locations (x_i, y_i)
- ▶ radii r_i .

What did we learn from the example?

- ▶ Towers with very small range could not activate others. So **large range** is helpful.
- ▶ Towers can only **directly** activate towers whose locations are inside their range. So maybe location is also important?
- ▶ A tower i can **indirectly** activate another tower j if there is a sequence of activations from i to j . Is there a way to identify when one tower can indirectly activate another?

Tsunami Warning

Attempt 1 (only consider range)

Find the unactivated tower with the largest range (breaking ties arbitrarily), and place a sensor at this tower. Find and remove all towers activated as a result. Repeat.

Attempt 2 (only consider direct activations)

Find the unactivated tower with the largest number of towers within its range (breaking ties arbitrarily), and place a sensor at this tower. Find and remove all towers activated as a result. Repeat.

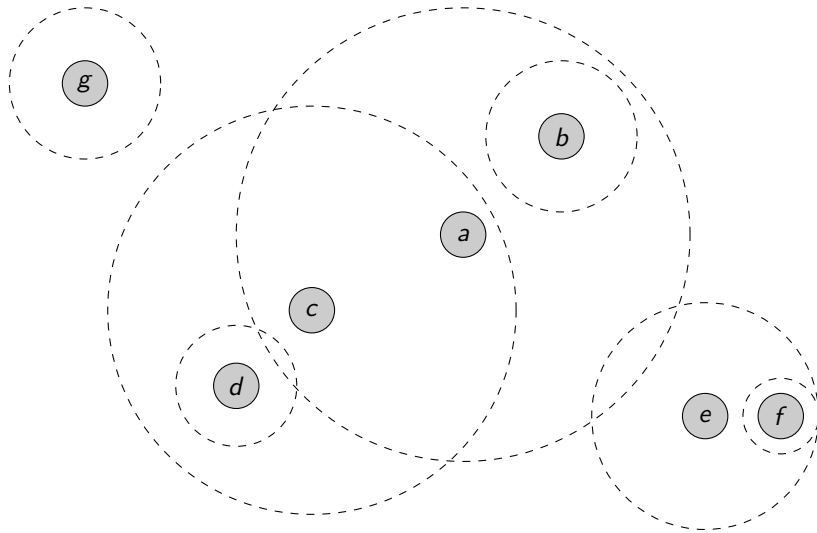
Exercise

Give examples which show that neither of these algorithms solve the problem correctly.

Our greedy algorithm also needs to consider indirect activations. How?

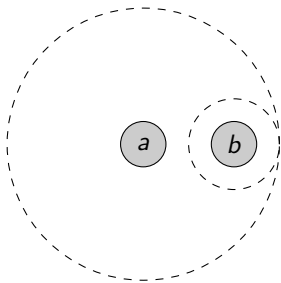
Tsunami Warning

Maybe we can use a graph?

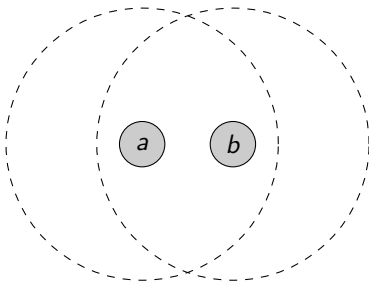


Tsunami Warning

It is useful to consider the towers as vertices of a directed graph, where an edge from tower a to tower b indicates that the activation of a **directly** causes the activation of b , that is, b is within the range of a .



$a \rightarrow b$



$a \leftrightarrow b$

Observation

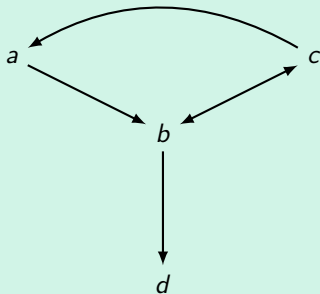
Suppose that activating tower a causes tower b to also be activated, and vice versa. Then we never want to place sensors at both towers; indeed, placing a sensor at a is equivalent to placing a sensor at b .

How can we extend this notion to a larger number of towers?

- ▶ Cycles also have this property.
- ▶ What other directed graphs have this property?

Tsunami Warning

Example



All four towers can be activated by placing just one sensor at a or b or c .

Observation

Let S be a subset of the towers such that that activating **any** tower in S causes the activation of **all** towers in S .

We never want to place more than one sensor in S , and if we place one, then it doesn't matter where we put it.

In this way, we can treat all of S as a unit; a "super-tower".

Strongly Connected Components

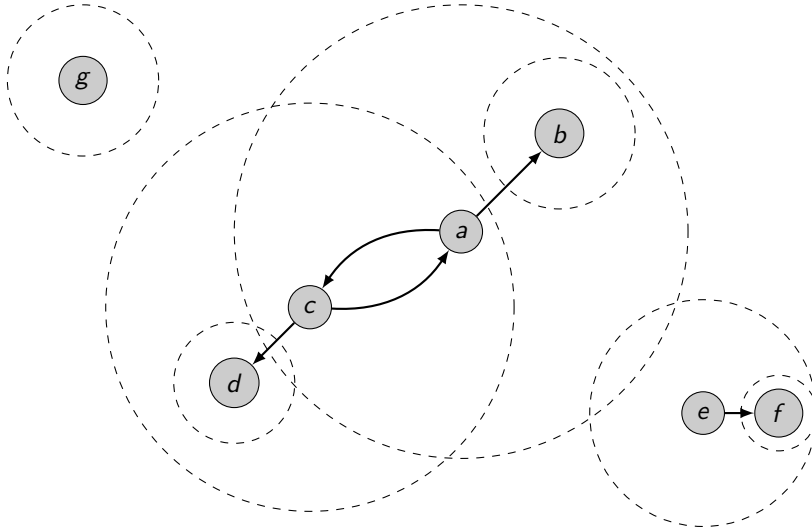
Definition (strongly connected component (SCC))

Given a directed graph $G = (V, E)$ and a vertex v , the **strongly connected component** (SCC) of G containing v consists of all vertices $u \in V$ such that there is a directed path in G from v to u and a directed path from u to v . We will denote it by C_v .

In the terms of our problem, strongly connected components are **maximal** super-towers.

Strongly Connected Components

What are the strongly-connected components below?



Strongly Connected Components

How do we find the strongly connected component $C_v \subseteq V$ containing v ?

Claim

u is in C_v if and only if u is reachable from v and v is reachable from u .

We can split this into two subproblems:

- ▶ Find all the vertices that can be reached from v .
- ▶ Find all the vertices which can reach v .

Then a vertex u is in C_v if and only if it appears in both of these sets.

Strongly Connected Components

To find all the vertices that can be reached from v :

- ▶ Starting at v , do a BFS or DFS of G ; vertex u is found in this search iff there is a path $v \rightsquigarrow u$ in G .

Finding the vertices that can reach v is harder:

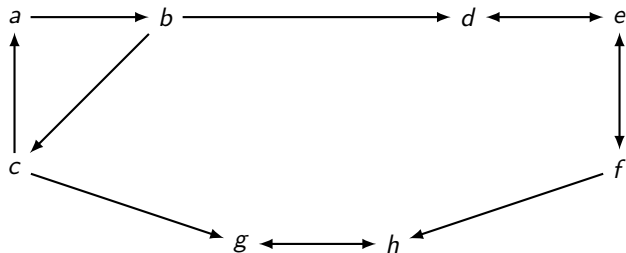
- ▶ Construct another graph $G_{rev} = (V, E_{rev})$ consisting of the same set of vertices V but with the set of edges E_{rev} obtained by reversing the direction of all edges E of G .
- ▶ Starting at v , do a BFS or DFS of G_{rev} ; vertex u is found in this search iff there is a path $v \rightsquigarrow u$ in G_{rev} , i.e., iff there is a path $u \rightsquigarrow v$ in G .

Claim

Vertex u is in C_v if and only if u is reachable from v and v is reachable from u .
In other words, u is reachable from v in both G and G_{rev} .
Equivalently, u is in both R_v and R'_v .

Strongly Connected Components

Suppose the original graph $G = (V, E)$ is

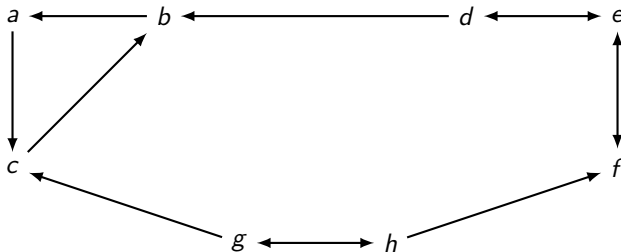


Then the set of vertices reachable from e is

$$R_e = \{d, e, f, g, h\}.$$

Strongly Connected Components

The reverse graph $G_{rev} = (V, E_{rev})$ is



Then the set of vertices reachable from e is

$$R'_e = \{a, b, c, d, e, f\}.$$

Strongly Connected Components

- ▶ Combining

$$R_e = \{d, e, f, g, h\}$$

with

$$R'_e = \{a, b, c, d, e, f\},$$

we have

$$C_e = R_e \cap R'_e = \{d, e, f\}.$$

- ▶ Note that C_d and C_f are also the same set, namely $\{d, e, f\}$.
- ▶ Similarly, we find that $C_a = C_b = C_c = \{a, b, c\}$ and $C_g = C_h = \{g, h\}$.

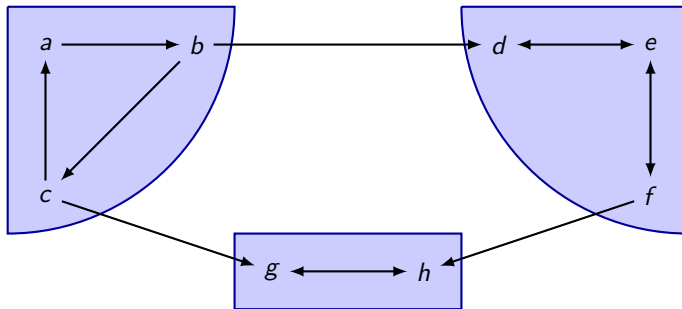
Strongly Connected Components

To recap:

- ▶ Use BFS to find the set $R_v \subseteq V$ of all vertices in V which are reachable in G from v .
- ▶ Similarly find the set $R'_v \subseteq V$ of all vertices which are reachable in G_{rev} from v .
- ▶ The strongly connected component of G containing v is given by $C_v = R_v \cap R'_v$.

Strongly Connected Components

Therefore the decomposition of G into strongly connected components is



Strongly Connected Components

Time complexity.

- ▶ Finding all strongly connected components in this way could require $O(|V|)$ traversals of the graph.
- ▶ Each of these traversals is a BFS, requiring $O(|V| + |E|)$ time.
- ▶ Therefore, the total time complexity is $O(|V| \cdot (|V| + |E|))$.
- ▶ Can we do better?
- ▶ Could one DFS suffice?

Tarjan's Algorithm (sketch)

- ▶ Do a DFS with an additional stack.
- ▶ Each vertex v also maintains
 - an id between 0 and $n - 1$, assigned in the order the vertices are encountered by the DFS,
 - a low value, which is the smallest id of a vertex that can be reached from v using DFS edges plus **one** back-edge, and
 - a Boolean `in-stack` which is true if the vertex is on the stack.
- ▶ When visiting a new vertex, push it onto the stack.
- ▶ When we encounter a back-edge (u, v) to a vertex v that is `in-stack`, we know that each vertex after v on the stack
 - can be reached from v (following the edges of the DFS), and
 - can reach v (using the edge we just encountered),

so they all belong to the same SCC as v . Vertex u updates its low value to $v.id$.

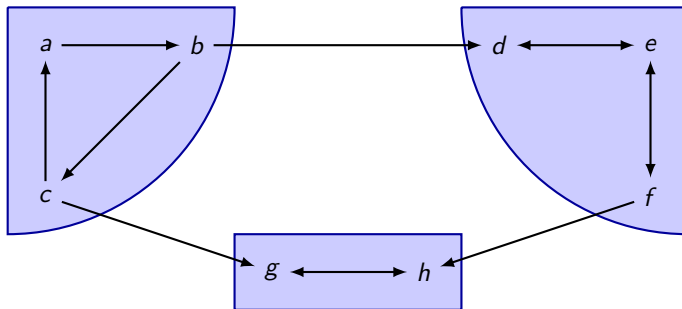
- ▶ When backtracking from a child c to a parent p in the DFS tree, the low values are propagated by setting
$$p.low = \min\{p.low, c.low\}.$$
- ▶ After having visited all neighbours of a vertex v , if $v.low = v.id$, then pop all vertices from the stack up to and including v ; these form one strongly connected component.
In fact, the condition $v.low = v.id$ tells us that these vertices cannot reach any other vertices.
- ▶ The time complexity is $O(|V| + |E|)$.

Strongly Connected Components

- ▶ Kosaraju's algorithm (presented in CLRS §22.5) is an alternative.
 - It involves two executions of DFS.
 - More complicated to reason about, easier to implement in code.
- ▶ It also runs in **linear time**, i.e., $O(|V| + |E|)$.
- ▶ Note that the input graph $G = (V, E)$ has size $|V| + |E|$. So linear time in the input size means linear in $|V| + |E|$.
- ▶ A linear time algorithm is asymptotically “no slower than” reading the graph, so we can run these algorithms “for free”, i.e., without worsening the time complexity of our solution to **any** problem.

Strongly Connected Components

But what should we do next?



- ▶ We've found a decomposition into strongly connected components.
- ▶ How do we use this to choose where to place the sensors?

The Condensation Graph

- ▶ It should be clear that distinct strongly connected components are disjoint sets, so the strongly connected components form a partition of V .
- ▶ Let C_G be the set of all strongly connected components of a graph G .

Definition (condensation graph)

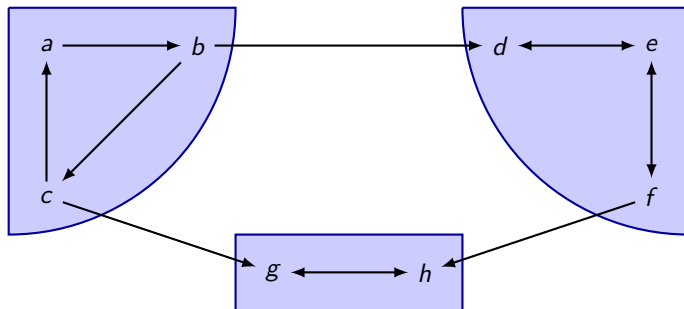
Define the **condensation graph** $\Sigma_G = (C_G, E^*)$, where

$$E^* = \{(C_u, C_v) : (u, v) \in E, C_u \neq C_v\}.$$

The vertices of Σ_G are the strongly connected components of G , and the edges of Σ_G correspond to those edges of G that are not within a strongly connected component, with duplicates ignored.

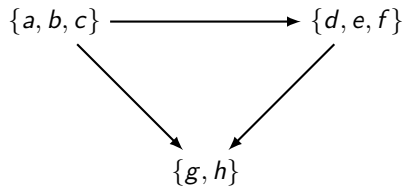
The Condensation Graph

Recall our earlier example



The Condensation Graph

The condensation graph is simply



For this tsunami problem instance,

- ▶ where would we place sensors on the condensation graph?
- ▶ Why?

The Condensation Graph

Let's recap:

- ▶ We begin our solution to the tsunami warning problem by finding the condensation graph.
- ▶ Now we have the set of super-towers (strongly connected components), and we know for each super-tower (SCC) which others it can activate.
- ▶ Our task is to decide which super-towers need a sensor installed in order to activate all the super-towers.
- ▶ We need to know one more property about the condensation graph.

Claim

The condensation graph Σ_G is a directed acyclic graph.

Proof Outline.

Suppose there is a cycle in Σ_G . Then the vertices on this cycle are not **maximal** SCCs, as they can be merged into an even larger SCC. □

Tsunami Warning

Solution

The correct greedy strategy is to only place a sensor in each super-tower without incoming edges in the condensation graph.

Proof of correctness.

These super-towers cannot be activated by another super-tower, so they each require a sensor. This shows that there is no solution using fewer sensors.

We still have to prove that this solution activates all super-towers.

Consider a super-tower with one or more incoming edges. Follow any of these edges backwards, and continue backtracking in this way.

Since the condensation graph is acyclic, this path must end at some super-tower without incoming edges. The sensor placed here will then activate all super-towers along our path.

Therefore, all super-towers are activated as required. □

Tsunami Warning

The steps we used to solve the tsunami problem were:

Algorithm

- ▶ **Input:** towers i with locations (x_i, y_i) and radii r_i .
- ▶ Construct a directed graph G , where (u, v) is an edge if tower u can directly activate tower v .
- ▶ Construct the reversed graph G_{rev} of G .
- ▶ Use DFS or BFS on G and G_{rev} to find the strongly connected components of G .
- ▶ Form the condensation graph Σ_G , whose vertices are the strongly connected components of G .
- ▶ **Solution:** Place a sensor on every vertex of Σ_G that has no incoming edges.

Tsunami Warning

Why is this a greedy algorithm? Here's one answer...

- ▶ We choose to place a sensor on a vertex v whose strongly-connected component C_v has the smallest number of incoming edges in Σ_v .
- ▶ We then delete v and all vertices that are reachable from v and repeat.
- ▶ It just so happens that this smallest number of incoming edges is always 0.

A more compelling answer is that we place a sensor on the first vertex of a topological ordering of the condensation graph, defined next.

Topological Ordering

In the special case where a directed graph G has no cycles (e.g., when G is a condensation graph), then we can order the vertices of G by the following rule:

Definition (topological ordering)

Let $G = (V, E)$ be a directed graph, and let $n = |V|$. A **topological ordering** of G is a linear ordering (permutation) of its vertices $\sigma : V \rightarrow \{1, \dots, n\}$ such that if there exists an edge $(v, w) \in E$ then v precedes w in the ordering, i.e., $\sigma(v) < \sigma(w)$.

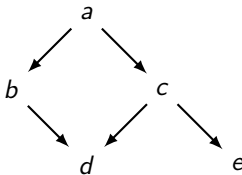
Property

Every directed acyclic graph has a topological ordering of its vertices.

Note: a topological ordering is not necessarily unique, i.e., there may be more than one valid topological ordering of the vertices.

Topological Ordering

Example. Find a topological ordering of the vertices in this graph:



There are many solutions, such as:

- ▶ *abcde*
- ▶ *acebd*

Why is *abdce* not a solution? This is because $\overrightarrow{cd}$ is an edge, but *d* appears before *c* in the list.

Kahn's Algorithm

How do we topologically order the vertices?

Algorithm

Maintain:

- ▶ a list L of vertices, initially empty,
- ▶ an array D consisting of the in-degrees of the vertices, and
- ▶ a set S of vertices with no incoming edges.

While set S is non-empty, select a vertex u of with $D[u] = 0$ in the set.

- ▶ Remove it from S and append it to L .
- ▶ Then, for every outgoing edge $e = (u, v)$ from this vertex, remove the edge from the graph, and decrement $D[v]$ accordingly.
 - If $D[v]$ is now zero, insert v into S .

If there are no edges remaining, then L is a topological ordering. Otherwise, the graph has a cycle.

Kahn's Algorithm

Time complexity.

- ▶ This algorithm runs in $O(|V| + |E|)$, that is, **linear time**.
- ▶ Once again, we can run this algorithm “for free” as it is asymptotically no slower than reading the graph.
- ▶ In problems involving directed acyclic graphs, it is often useful to impose a topological ordering and then think about the actual problem!
- ▶ A topological ordering is often a natural way to process the vertices. We'll see more of this in Dynamic Programming.

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Single Source Shortest Paths

Problem

Instance: a directed graph $G = (V, E)$ with **non-negative** edge-weights $w(e)$, and a designated **source** vertex $s \in V$, such that for every $v \in V$ there is a path from s to v .

Task: for each $v \in V$, find the smallest weight of a path from s to v .

Example

Suppose $G = (V, E)$ models Sydney's road network with roads E and intersections V . The weight of an edge $\overrightarrow{uv}$ records how much time it takes to drive down the road from u to v . Your home is located at intersection s , and you want to identify the quickest journey from your home to any other location in Sydney.

Observe that this problem makes the following assumptions:

- ▶ There is exactly one source vertex s .
- ▶ The graph is directed.
- ▶ The edge weights are non-negative.

There are other versions of this problem that make different assumptions; we'll see some of them later in the course.

Note

To find shortest paths from s in an **undirected** graph, simply replace each undirected edge with two directed edges in opposite directions.

Note

There isn't necessarily a unique shortest path from s to each vertex.

Dijkstra's Algorithm

This task is accomplished by a very elegant greedy algorithm developed by Edsger Dijkstra in 1959.

Algorithm Outline

Maintain a set S of vertices for which the shortest path weight has been found, initially empty. S is represented by a boolean array.

For every vertex v , maintain a value d_v which is the weight of the shortest 'known' path from s to v , i.e., the shortest path using only intermediate vertices in S . Initially $d_s = 0$ and $d_v = \infty$ for all other vertices.

At each stage, we greedily add to S the vertex $v \in V \setminus S$ which has the smallest d_v value. Record this value as the length of the shortest path from s to v , and update other d_z values as necessary.

Dijkstra's Algorithm

This outline still leaves much work to do. We will examine the following four features of Dijkstra's algorithm:

- ▶ **Correctness.** Why is it correct to always add the vertex outside S with the smallest d_v value?
- ▶ **Updates.** When v is added to S , for which vertices z must we update d_z , and how do we do these updates?
- ▶ **Data structure.** What data structure should we use to represent the d_v values?
- ▶ **Time complexity.** What is the time complexity of this algorithm, and how is it impacted by our choice of data structure?

Dijkstra's Algorithm: correctness

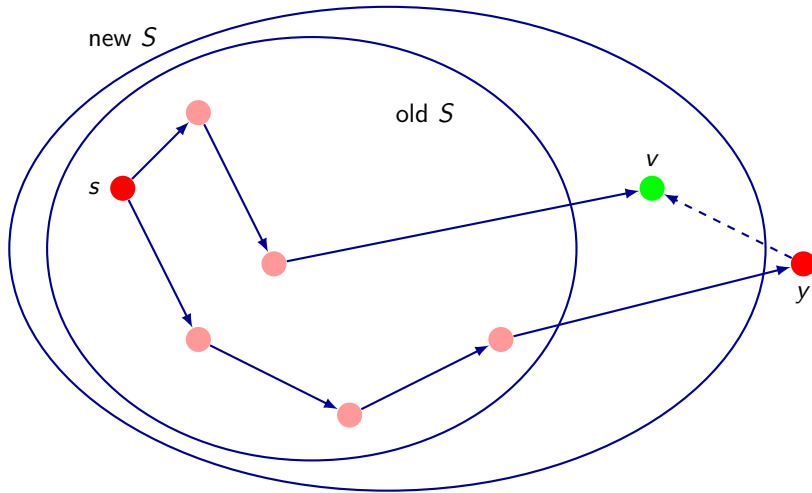
Claim

Suppose v is the next vertex to be added to S . Then d_v is the length of the shortest path from s to v .

Proof (1/2).

- ▶ d_v is the length of a shortest path from s to v using only intermediate vertices in S . Let's call this path p .
- ▶ If this were **not** a shortest path from s to v , there must be some shorter path p' which first leaves S at some vertex y before later reaching v .

Dijkstra's Algorithm: correctness



Dijkstra's Algorithm: correctness

Proof (2/2).

- ▶ Now, the portion of p' up to y is a path from s to y using only intermediate vertices in S .
- ▶ Therefore, this portion of p' has weight at least d_y .
- ▶ Since all edge weights are non-negative, p' itself has weight at least d_y .
- ▶ But v was chosen to have smallest d -value among all vertices outside S !
- ▶ So we know that $d_v \leq d_y$, and hence the weight of path p is at most that of p' .
- ▶ Therefore, d_v is indeed the weight of a shortest path from s to v .



Dijkstra's Algorithm: updates

Question

Earlier, we said that when we add a vertex v to S , we may have to update some d_z values. What updates could be required?

Answer

If there is an edge from v to z with weight $w((v, z))$, the shortest known path to z may be improved by taking the shortest path to v followed by this edge. Therefore, we check whether

$$d_z > d_v + w((v, z)),$$

and if so we update d_z to the value $d_v + w((v, z))$.

As it turns out, these are the **only** updates we should consider!

Dijkstra's Algorithm: updates

Claim

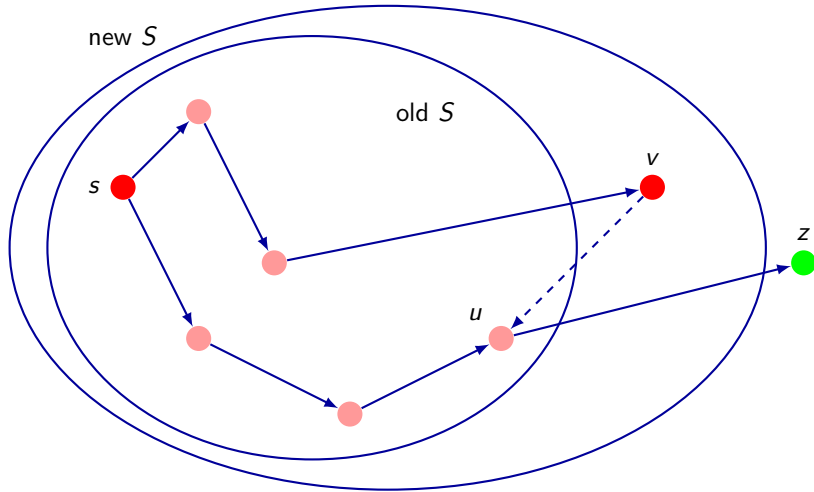
If d_z changes as a result of adding v to S , the new shortest known path to z must have penultimate vertex v , i.e., the last edge must go from v to z .

Proof (1/2).

- ▶ Suppose that adding v to S allows for a new shortest path through S from s to z with penultimate vertex $u \neq v$.
- ▶ Such a path must include v , or else it would not be new. Thus the path is of the form

$$p = s \rightarrow \cdots \rightarrow v \rightarrow \cdots \rightarrow u \rightarrow z.$$

Dijkstra's Algorithm: updates



Dijkstra's Algorithm: updates

Proof (2/2).

- ▶ Since u was added to S before v was, we know that there is a shortest path p' from s to u which does not pass through v .
- ▶ Appending the edge $\vec{uz}$ to p' produces a path through S from s to z which is no longer than p .
- ▶ This path was already a candidate for d_z , so the weight of p is greater than or equal to the existing d_z value.
- ▶ This is a contradiction, so the proof is complete.



Dijkstra's Algorithm: data structures

- ▶ Now, we are ready to consider data structures to maintain the d_v values.
- ▶ We need to support two operations:
 - find the vertex $v \in V \setminus S$ with smallest d_v value, and
 - for each of its outgoing edges (v, z) , update d_z if necessary.
- ▶ We'll start with the simplest data structure: the array.

Dijkstra's Algorithm: array

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

Suppose the vertices of the graph are labelled $1, \dots, n$.

Recall that vertex s is the source.

Attempt 1

Store the d_i values in an array $d[1..n]$. At each stage:

- ▶ Perform a linear search of array d , ignoring those vertices already in S , and select the vertex v with smallest $d[v]$ to be added to S .
- ▶ For each outgoing edge from vertex v to some $z \in V \setminus S$, update $d[z]$ if necessary.

Question

What is the time complexity of this algorithm?

Answer

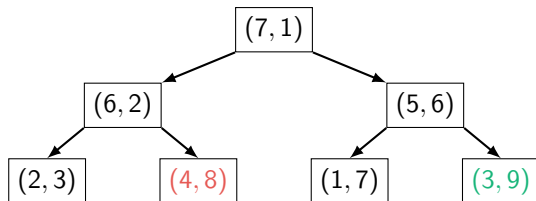
- ▶ At each of n steps, we perform a linear scan on an array of length n .
- ▶ We also run the update procedure (in constant time) at most once for each edge.
- ▶ The algorithm therefore runs in $O(n^2 + m) = O(n^2)$ time.
- ▶ If the graph is **dense** (i.e., $m = \Theta(n^2)$), this is best possible.
- ▶ Can we do better when $m \ll n^2$?
- ▶ For example, road networks are not dense, and have $m = O(n)$.

Dijkstra's Algorithm: data structures

- ▶ Recall the two operations we need to support:
 - find the vertex $v \in V \setminus S$ with smallest d_v value, and
 - for each of its outgoing edges (v, z) , update d_z if necessary.
- ▶ So far, we have done the first operation in $O(n)$ using a linear search.
- ▶ How can we improve on this?
- ▶ The first operation isn't a pure 'find minimum' because we have to skip over vertices already in S .
- ▶ Instead, when we add a vertex v to S , we could try deleting d_v from the data structure altogether.
- ▶ We now have three operations to support:
 - find minimum,
 - delete minimum, and
 - update any.
- ▶ The first two of these suggest the use of a min-heap, but the standard heap doesn't allow us to update arbitrary elements.

Augmented Heaps

- ▶ We use a heap represented by an array $A[1..n]$; the left child of $A[j]$ is stored in $A[2j]$ and the right child in $A[2j + 1]$.
- ▶ Every element of A is of the form $A[j] = (i, d_i)$ for some vertex i . The min-heap property is maintained with respect to the d -values only.
- ▶ We also maintain another array $P[1..n]$ which stores the **position** of elements in the heap.
- ▶ Whenever $A[j]$ refers to vertex i , we record $P[i] = j$, so that we can look up vertex i using the property $A[P[i]] = (i, d_i)$.



j	1	2	3	4	5	6	7	
$A[j]$	7	6	5	2	4	1	3	i
	1	2	6	3	8	7	9	d_i

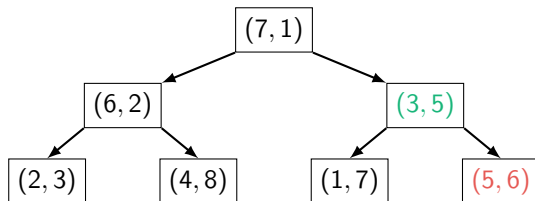
i	1	2	3	4	5	6	7
$P[i]$	6	4	7	5	3	2	1

Augmented Heaps

- ▶ Updating the d -value of vertex i is now an $O(\log n)$ operation.
 - First, look up vertex i in the position array P . This gives us $P[i]$, the index in A where the pair (i, d_i) is stored.
 - Next, we update d_i by changing the second entry of $A[P[i]]$.
 - Finally, it may be necessary to bubble up or down to restore the min-heap property. In this algorithm, d -values are only ever reduced, so only bubbling up is applicable.
- ▶ Accessing the top of the heap still takes $O(1)$ time, and popping the heap still takes $O(\log n)$ time.

Example. Update d_3 from 9 to 5.

After the update:



j	1	2	3	4	5	6	7	
$A[j]$	7	6	3	2	4	1	5	i
	1	2	5	3	8	7	6	d_i

i	1	2	3	4	5	6	7
$P[i]$	6	4	3	5	7	2	1

Dijkstra's Algorithm: augmented heap

Algorithm

Store the d_i values in an augmented heap of size n . At each stage:

- ▶ Access the top of the heap to obtain the vertex v with smallest key and add it to set S .
- ▶ Pop the corresponding element d_v from the heap.
- ▶ For each outgoing edge from v to some $z \in V \setminus S$, update d_z if necessary.

Dijkstra's Algorithm: Augmented Heap

Question

What is the time complexity of our algorithm?

Answer

- ▶ Build the initial heap in $O(n \log n)$ time (faster algorithms also exist).
- ▶ Each of n stages requires a deletion from the heap (when a vertex is added to S), which takes $O(\log n)$ steps.
- ▶ Each edge causes at most one update of a key in the heap, also taking $O(\log n)$ steps.
- ▶ Thus, in total, the algorithm runs in time $O((n + m) \log n)$.
- ▶ But since there is a path from v to every other vertex, we know that $m \geq n - 1$, so we can simplify to $O(m \log n)$ total time.

Dijkstra's Shortest Paths Algorithm

Note

In COMP2521/9024, you may have seen that the time complexity of Dijkstra's algorithm can be improved to $O(m + n \log n)$. This is true, but it relies on an advanced data structure called the **Fibonacci heap**, which has not been taught in this course or any prior course.

Therefore, this improvement will be considered **external** to our course. You cannot use it in assessments.

Table of Contents

1. Motivation

2. Optimal Selection

2.1 Activity selection

2.2 Cell tower placement

3. Optimal Ordering

3.1 Tape storage

3.2 Minimising job lateness

4. Optimal Merging

4.1 The Huffman Code

5. Applications to graphs

5.1 Directed graph structure

5.2 Single source shortest paths

5.3 Minimum spanning trees

Minimum Spanning Trees

Definition (spanning tree, weight of a spanning tree, minimum spanning tree)

$G = (V, E)$ is a connected graph and each edge e in E has a non-negative length.

- ▶ A **spanning tree** T of G is any tree which is a subgraph of G on the same vertex set V .
- ▶ The **weight** of a tree T is the sum of all edge lengths in T .
- ▶ A **minimum spanning tree** T of a connected graph G is a spanning tree of G with the smallest weight.

Lemma

Let $G = (V, E)$ be a connected graph with:

- ▶ all lengths of edges distinct, and
- ▶ S a non-empty proper subset of V .

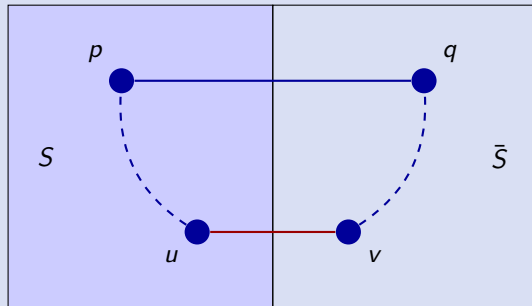
Let $e = \{u, v\}$ be a shortest edge between S and $V \setminus S$ (w.l.o.g., $u \in S$ and $v \notin S$).
Then e belongs to every minimum spanning tree T of G .

Minimum Spanning Trees

Proof.

Assume that there is a minimum spanning tree T which does not contain the edge $e = \{u, v\}$.

- ▶ Since T is a spanning tree, there is a path from u to v within T , and this path must leave S by some edge, say $\{p, q\}$ where $p \in S$ and $q \notin S$.
- ▶ However, $\{u, v\}$ is shorter than any other edge with one end in S and one end outside S , including $\{p, q\}$.
- ▶ Replacing the edge $\{p, q\}$ with the edge $\{u, v\}$ produces a new tree T' with smaller total edge weight.
- ▶ This contradicts our assumption that T is a minimum spanning tree, completing the proof.



Minimum Spanning Trees

- ▶ There are two famous greedy algorithms for the minimum spanning tree problem.
- ▶ Both algorithms build up a forest, beginning with all n isolated vertices and adding edges one by one.
- ▶ **Prim's algorithm** uses one large component, adding one of the isolated vertices to it at each stage. This algorithm is very similar to Dijkstra's algorithm, but adds the vertex closest to S rather than the one closest to the starting vertex v .
- ▶ We will instead focus on **Kruskal's algorithm**.

Kruskal's Algorithm

- ▶ Sort the edges E in increasing order by weight.
- ▶ An edge e is added if its inclusion does not introduce a cycle in the graph constructed thus far, or discarded otherwise.
- ▶ The process terminates when the forest is connected, i.e., when $n - 1$ edges have been added.

Claim

Kruskal's algorithm produces a minimum spanning tree, and if all weights are distinct, then such a minimum spanning tree is unique.

Partial proof (all weights are distinct).

- ▶ Consider an edge $e = \{u, v\}$ added by Kruskal's algorithm, and let F be the forest **before** adding e .
- ▶ Let S be the set of vertices reachable from u in F . Then, $u \in S$ but $v \notin S$.
- ▶ The graph has no edge shorter than e with one endpoint in S and the other outside S . If such an edge existed, it would have been considered before e .
- ▶ Since e is the shortest edge between S and $\bar{S}$ it belongs to every minimum spanning tree, by the previous lemma.
- ▶ Thus, the edges produced by Kruskal's algorithm are contained in every minimum spanning tree.
- ▶ But the graph produced by Kruskal's algorithm by definition:
 - has no cycles,
 - is connected (so it is a tree),
 - contains all the vertices of G (so it is a spanning tree)
- ▶ Therefore, when all edge weights are distinct, Kruskal's algorithm produces the unique minimum spanning tree.

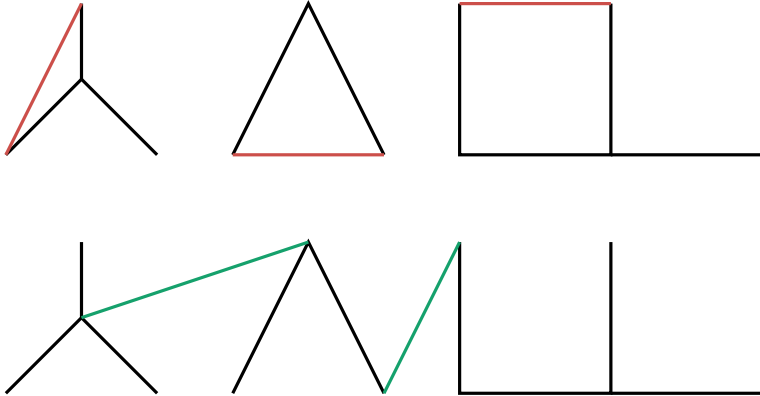


Kruskal's Algorithm: efficient implementation

Kruskal's algorithm

- ▶ We sort the edges E in increasing order by weight.
 - ▶ An edge e is added if its inclusion does not introduce a cycle in the graph constructed thus far, or discarded otherwise.
 - ▶ The process terminates when the forest is connected, i.e., when $n - 1$ edges have been added.
-
- ▶ To efficiently implement Kruskal's algorithm, we need to quickly determine whether a certain new edge will introduce a cycle.
 - ▶ An edge $e = (u, v)$ will introduce a cycle in the forest F if and only if there is already a path between u and v , i.e., u and v are in the same connected component.

Kruskal's Algorithm: efficient implementation



In our implementation of Kruskal's algorithm, we store the vertices using the **Union-Find** data structure. It handles disjoint sets, supporting the following operations:

1. $\text{UNION}(a, b)$ merges the set A containing a and the set B containing b into a single set $A \cup B$.
2. $\text{FIND}(a)$ returns the representative of the set containing a .
 - $\text{FIND}(a) = \text{FIND}(b)$ if and only if a and b are in the same set.
 - $\text{FIND}(a)$ can change over time, as a result of UNION operations.

Idea

Maintain a forest, where each tree represents one of our disjoint sets.

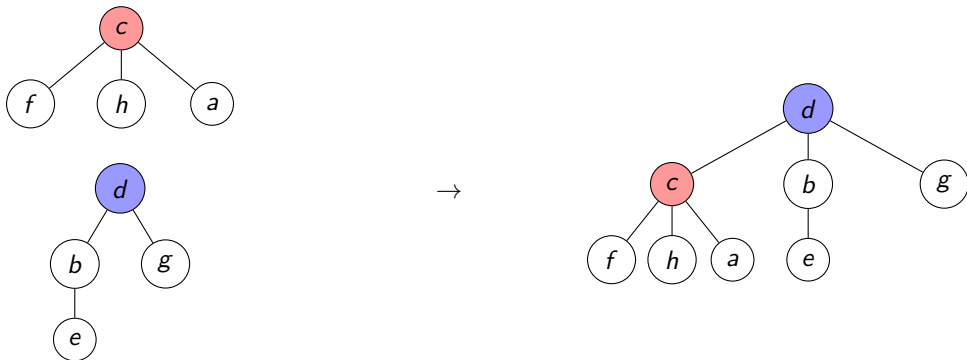
- ▶ Begin with n vertices and no edges, so initially each vertex belongs to a different set.
- ▶ What representative should the `FIND` operation return?
 - Keep each tree rooted, and return the root.
- ▶ How to maintain the rooted structure?
Use an array:
 - non-root vertices record their parent, and
 - root vertices record a flag to indicate that they have no parent.

Union-Find

- ▶ $\text{FIND}(a)$ should find the root of the tree containing vertex a .
 - Starting at a , repeatedly traverse the edge to the parent, until there is no parent.
- ▶ $\text{UNION}(a, b)$ should:
 - do nothing if a and b are already in the same tree, or
 - add an edge otherwise, to merge the two trees into one.

Union-Find

We place the edge between the roots of the trees containing a and b .



- ▶ In the example, we made d (the root of b 's tree) become the parent of c (the root of a 's tree).
- ▶ We could have just as easily made c become the parent of d .
- ▶ We'll later see how to make this choice more systematically!

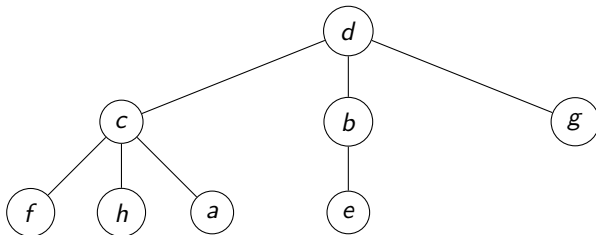
Union-Find

In summary, we process $\text{UNION}(a, b)$ as follows.

1. Run $\text{FIND}(a)$ and $\text{FIND}(b)$ to identify the roots of each tree, say c and d respectively.
 2. If $c = d$, then do nothing.
 3. Otherwise, place an edge between c and d , choosing (for now) arbitrarily which will become the parent of the other.
-
- ▶ A FIND operation takes $O(h)$ time, where h is the height of the tree being traversed.
 - ▶ A UNION operation involves two FIND operations and potentially one edit to the parent array, so this also takes $O(h)$ time.
 - ▶ What height can the trees have?
 - If we don't choose wisely when merging two trees together, we may still create a very unbalanced tree, where $h \approx n$.
 - Fortunately, a simple change brings h (and therefore both operations) down to $O(\log n)$.

Union-Find: size heuristic

For each vertex, maintain not only the parent, but also the size of the subtree rooted at that vertex.



v	a	b	c	d	e	f	g	h
$\text{parent}(v)$	c	d	d	—	b	c	d	c
$\text{subtree}(v)$	1	2	4	8	1	1	1	1

Union-Find: size heuristic

Process $\text{UNION}(a, b)$ as follows.

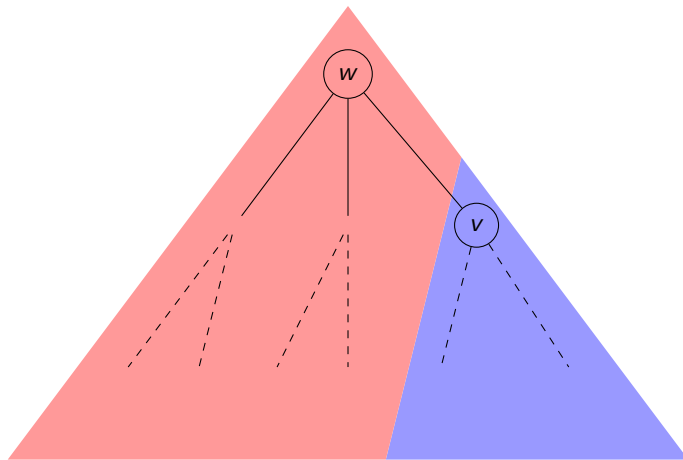
1. Run $\text{FIND}(a)$ and $\text{FIND}(b)$ to identify the roots of each tree, say c and d respectively.
2. If $c = d$, then do nothing.
3. Otherwise, place an edge between c and d :
 - if $\text{subtree}(c) \geq \text{subtree}(d)$,¹ then set $\text{parent}(d) = c$, or
 - if $\text{subtree}(d) > \text{subtree}(c)$, then set $\text{parent}(c) = d$.

¹The case $\text{subtree}(c) = \text{subtree}(d)$ can be handled arbitrarily; we can set either c or d to be the parent of the other.

Union-Find: size heuristic

- ▶ What does this have to do with tree height?
- ▶ **Key property:** once a vertex v gets a parent w , nothing will be added to v 's subtree in the future.
- ▶ Just before we placed this edge, $\text{subtree}(w)$ was already $\geq \text{subtree}(v)$.
- ▶ Placing this edge put v 's subtree under w , thus increasing $\text{subtree}(w)$ by $\text{subtree}(v)$.
- ▶ The new value of $\text{subtree}(w)$ is therefore $\geq 2 \text{subtree}(v)$.
- ▶ As we continue to place edges, $\text{subtree}(w)$ might continue increasing, but $\text{subtree}(v)$ will not change.

Union-Find: size heuristic



Union-Find: size heuristic

Claim

The size heuristic guarantees that each tree in the data structure has height at most $\log_2 n$.

Proof.

We saw earlier that if w is the parent of v , then $\text{subtree}(w) \geq 2 \cdot \text{subtree}(v)$ **at all times**.

If the height of the tree is h , there is a path of h parent edges from some leaf vertex v to the root r . This is a chain of h parents.

As we traverse that path, the subtree size must at least double at each step. Therefore,

$$\begin{aligned}\text{subtree}(v) &= 1 \\ \text{subtree}(\text{parent}(v)) &\geq 2 \\ \text{subtree}(\text{parent}(\text{parent}(v))) &\geq 4 \\ &\dots \geq \dots \\ \text{subtree}(r) &\geq 2^h.\end{aligned}$$

However, there are only n vertices in the forest, so $\text{subtree}(r) \leq n$.

It follows that $2^h \leq n$, and therefore $h \leq \log_2 n$.



Union-Find: Size Heuristic

- ▶ A FIND operation takes $O(h)$ time, where h is the height of the tree being traversed.
- ▶ A UNION operation involves two FIND operations and potentially one edit to the parent array, so this also takes $O(h)$ time.
- ▶ The size heuristic guarantees that $h \leq \log_2 n$, so both operations now run in $O(\log n)$.
- ▶ This Union-Find data structure is good enough to get the sharpest possible bound on the complexity of Kruskal's algorithm.
- ▶ CLRS §19 presents an (equally good) alternative to the size heuristic, known as the **rank heuristic**, and extends it using **path compression** to further reduce the **amortised** complexity of both operations.

Efficient Implementation of Kruskal's Algorithm

- ▶ We now use the previously described Union-Find data structure to efficiently implement Kruskal's algorithm on a graph $G = (V, E)$ with n vertices and m edges.
- ▶ We first have to sort m edges of graph G which takes time $O(m \log m)$. Since $m < n^2$, we can rewrite this as $O(m \log n^2) = O(m \log n)$.
- ▶ As we progress through the execution of Kruskal's algorithm, we will start with n isolated vertices, which will be merged into connected components until all vertices belong to a single connected component. We use the Union-Find data structure to keep track of the connected components constructed at any stage.

Efficient Implementation of Kruskal's Algorithm

- ▶ For each edge $e = (u, v)$ on the sorted list of edges, we use two `FIND` operations to find their representatives x and y .
- ▶ If u and v do not belong to the same component (i.e., $x \neq y$), we add e to the spanning tree being constructed and perform `UNION`(u, v) to merge the trees containing u and v .
- ▶ If instead $x = y$, there is already a path between u and v , so adding this edge would create a cycle. Therefore, we simply discard e .
- ▶ In the worst case, we perform $2m$ `FIND` operations (checking the endpoints of each edge), each costing $O(\log n)$.
- ▶ We always perform $n - 1$ `UNION` operations (placing the edges of the MST), each costing $O(\log n)$.
- ▶ The overall time complexity is therefore $O(2m \log n + n \log n) = O((m + n) \log n)$.
- ▶ We can safely assume $m \geq n - 1$ (as otherwise there is no spanning tree), so this can be summarised further to $O(m \log n)$.
- ▶ Sorting the edges is a bottleneck.

Clustering of Maximum Spacing

Problem: Clustering of Maximum Spacing

Instance: A complete graph G with weighted edges representing distances between the vertices, and an integer k .

Task: Partition the vertices of G into k disjoint subsets so that the smallest distance between two points belonging to different sets of the partition is as large as possible.

Solution: partial Kruskal's algorithm

Sort the edges in increasing order and start performing the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k connected components, rather than a single spanning tree.

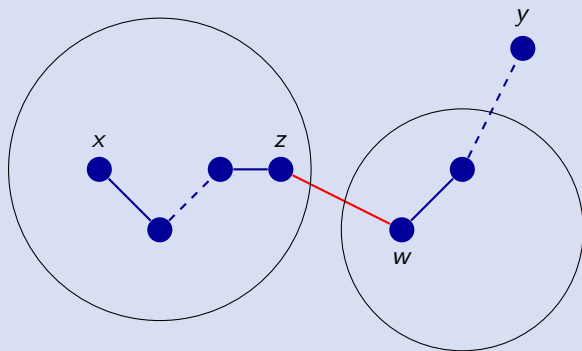
Clustering of Maximum Spacing

Proof of optimality (1/2).

- ▶ Suppose our clustering achieves a spacing of δ .
 - This means that every edge in the forest has weight at most δ .
- ▶ Consider any clustering different from the one produced by our algorithm. We will show that this other clustering is not better than ours.
- ▶ There must be some pair of vertices x and y that are clustered together in our algorithm but are separated in the other clustering.
- ▶ Since x and y belong to the same tree of **our** forest, there is a path p in that component connecting x and y . Every edge on this path was selected by the partial Kruskal's algorithm, so all these edges must have weight at most δ .

Proof of optimality (2/2).

- ▶ But in the other clustering (indicated by the circles in the diagram), y is not in the same cluster as x , so this $x \rightarrow y$ path p must leave x 's cluster at some edge $\{z, w\}$.
- ▶ $\{z, w\}$ is now an edge from one cluster to another, and its weight is at most δ , so the other clustering has spacing at most δ .
- ▶ Therefore, any other clustering is no better than ours!



k -clustering of maximum spacing

- ▶ What is the time complexity of this algorithm?
- ▶ We have $\Theta(n^2)$ edges; thus sorting them by weight will take $O(n^2 \log n^2)$, which we can simplify to $O(n^2 \log n)$.
- ▶ As usual, the running time of Kruskal's algorithm is dominated by sorting the edges.
- ▶ So the algorithm has time complexity $O(n^2 \log n)$ in total.



That's All, Folks!!