



UNSW  
SYDNEY

## 4. THE GREEDY METHOD

Aleks Ignjatović, [ignjat@cse.unsw.edu.au](mailto:ignjat@cse.unsw.edu.au)

office: K17 504

Course Admin: Song Fang, [cs3121@cse.unsw.edu.au](mailto:cs3121@cse.unsw.edu.au)

School of Computer Science and Engineering  
UNSW Sydney

Term 2, 2023

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

### 3.1 Directed graph structure

### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

## Question

What is a greedy algorithm?

## Answer

A greedy algorithm is one that solves a problem by doing it in stages, only considering the choice that appears to be the best at that stage of construction.

This obviously reduces the search space, but it works correctly only in cases when the locally optimal choices lead to the globally optimal outcome.

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.

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

### 3.1 Directed graph structure

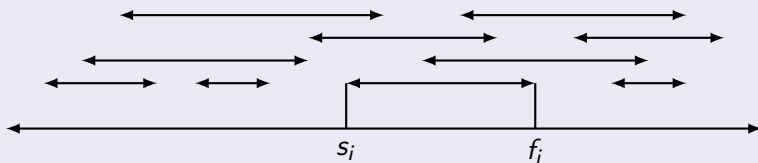
### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

## Problem

**Instance:** A list of  $n$  activities, with starting times  $s_i$  and finishing times  $f_i$ . No two activities can take place simultaneously.



**Task:** Find a *maximum size* subset of compatible activities.

## Attempt 1

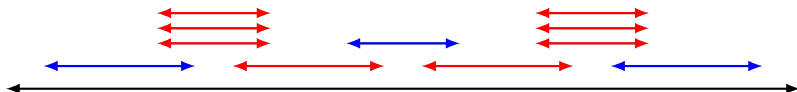
Always choose the shortest activity which does not conflict with the previously chosen activities, then remove the conflicting activities and repeat.



In the above example, our proposed algorithm chooses the activities in blue, then has to discard all the red activities, so clearly this does not work.

## Attempt 2

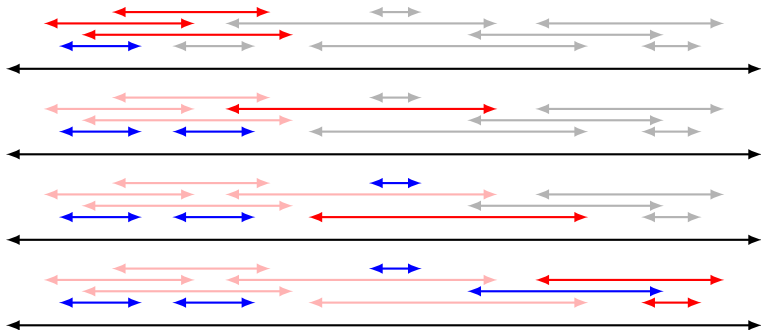
Maybe we should always choose an activity which conflicts with the fewest possible number of the remaining activities? It may appear that in this way we minimally restrict our next choice ...



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

## Solution

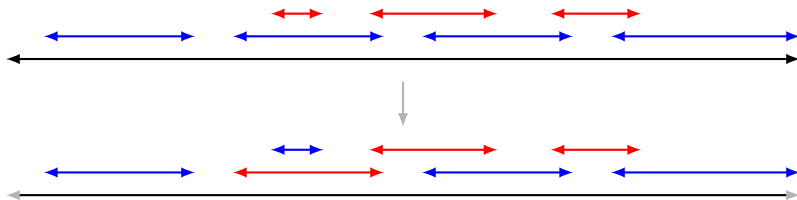
Among those activities which do not conflict with the previously chosen activities, always choose the activity with the earliest end time (breaking ties arbitrarily).



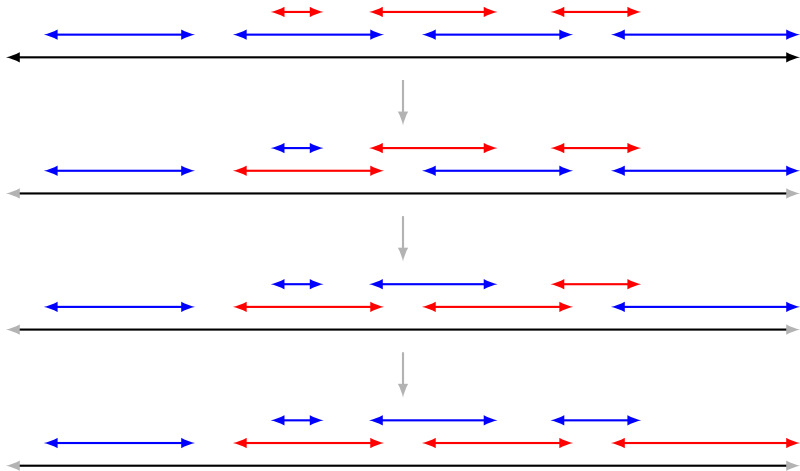


To prove the correctness of our algorithm, we will use an *exchange argument*. We will show that any alternative solution can be transformed into a solution obtained by our greedy algorithm with at least as many activities.

- Find the first place where the chosen activity violates the greedy choice.
- What if we replace that activity with the greedy choice?



- Does the new selection have any conflicts? No!
- Does the new selection have the same number of activities? Yes!
- So the greedy choice is actually just as good as the choice used in the alternative solution!
- We replace it and repeat.
- Continuing in this manner, we can eventually “morph” any alternative solution into the greedy solution, thus proving the greedy solution is optimal.



- 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), in  $O(n \log n)$  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  $i$ , starting at  $s_i$  and finishing at  $f_i$ , with all earlier finishing activities already 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.

- 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 to be 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.
- Thus, the algorithm runs in total time  $O(n \log n)$ , dominated by sorting.

## 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. No two activities can take place simultaneously.

**Task:** Find a subset of compatible activities of *maximal 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.

## Question

What happens if the activities are not all of the same duration and we have to select activities of maximal total duration?

## Solution

The greedy strategy no longer works - we will need a more sophisticated technique.



### Problem

**Instance:** You are traveling by car on a road from Loololong (in the West) to Goolagong (in the East). You start with full tank of petrol which you know that is sufficient to travel  $K$  kilometres. You also know the distances  $d_i$  from Loolongong to  $N$  petrol stations on the road and you wish to reach Goolagong with a minimal possible number of stops to refuel? How should you choose at which petrol stations to stop?

### Solution

Always travel to the furthest petrol station you can reach without running out of petrol and always fill the tank to its capacity.

- Why is this optimal?
- Clearly for any given number of stops, such greedy strategy allows you to travel the longest distance.
- So if there is a way to reach Goolagong with  $n$  stops, this is possible with the greedy strategy.
- Greedy solutions always stays ahead.

## Question

So, in general, how do we prove that a greedy algorithm produces a correct solution?

## Answer

There are two main methods of proof:

- 1 *Exchange argument*: consider an alternative solution, and gradually transform it to the solution found by our proposed algorithm without making it any worse (as we did in the Activity Selection Problem).
- 2 *Greedy stays ahead*: prove that at every stage, no other sequence of choices could do better than our proposed algorithm (as we did in the Petrol Stations Problem).

- When does Greedy strategy work?
- There is a theory dealing with this issue (*matroids*, presented in CLRS textbook) but it is not very practical.
- After solving a sufficient number of problems you will develop intuition when the Greedy strategy produces a correct solution.
- It is important that you always prove the correctness of the algorithm, because sometimes it is not obvious if Greedy Method produces a correct solution!

**Problem**

**Instance:** Along the long, straight road from Loololong (in the West) to Goolagong (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 minimal number of cell towers alongside the road, that is sufficient to cover all houses.

- Maybe we should put the first tower at the spot where it would cover the largest number of houses, then remove the covered houses and continue in this manner until all houses are covered.
- Exercise: Show that this does not work by finding a counter example.

- Let us attempt a different 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 should 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.

- 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.

### Exercise

Prove the correctness of this algorithm using an exchange argument.



- One of Telstra's engineers started with the house closest to Loololong and put a tower 5km away to the east. He then found the westmost house not already in the range of the tower and placed another tower 5 km to the east of it and continued in this way till he reached Goolagong.
- His junior associate did exactly the same but starting from Goolagong and moving westwards and claimed that his method required fewer towers.
- Is there a placement of houses for which the associate is right?

## Problem

**Instance:** A start time  $T_0$  and a list of  $n$  jobs, with duration times  $t_i$  and deadlines  $d_i$ . Only one job can be performed at any time; all jobs have to be completed. If a job  $i$  is completed at a finishing time  $f_i > d_i$  then we say that it has incurred lateness  $l_i = f_i - d_i$ .

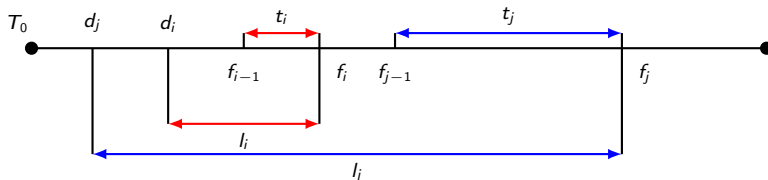
**Task:** Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

## Solution

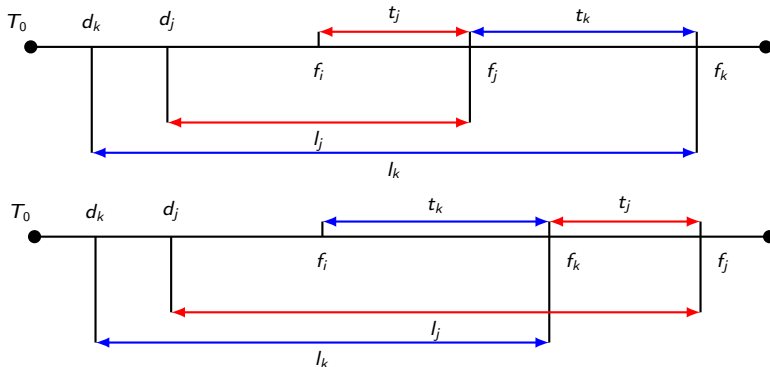
Ignore job durations and schedule jobs in the increasing order of deadlines.

## Proof of optimality

Consider any alternative schedule. We say that jobs  $i$  and  $j$  form an inversion if job  $i$  is scheduled before job  $j$  but  $d_j < d_i$ .



- We will show that there exists a scheduling without inversions which is at least as good.
- Recall that bubble sort only swaps adjacent array entries, and eventually sorts the array.
- Thus, if we show that we can swap two adjacent inverted activities without increasing the largest lateness then we are guaranteed that the schedule without any inversions is optimal.



Note that swapping adjacent inverted jobs reduces the larger lateness!

## 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.

- If the files are stored in order  $l_1, l_2, \dots, l_n$ , then the expected time is proportional to

$$\begin{aligned} & l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \dots + (l_1 + l_2 + l_3 + \dots + l_n) \\ &= nl_1 + (n-1)l_2 + (n-2)l_3 + \dots + 2l_{n-1} + l_n. \end{aligned}$$

- This is minimised if  $l_1 \leq l_2 \leq l_3 \leq \dots \leq l_n$ , so we simply sort the files by increasing order of length for an  $O(n \log n)$  solution.

## 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.



- If the files are stored in order  $l_1, l_2, \dots, l_n$ , then the expected time is proportional to

$$l_1 p_1 + (l_1 + l_2) p_2 + (l_1 + l_2 + l_3) p_3 \\ + \dots + (l_1 + l_2 + l_3 + \dots + l_n) p_n$$

- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio  $p_i/l_i$ .

- Let us see what happens if we swap two adjacent files, say files  $k$  and  $k + 1$ .
- The expected time before the swap and after the swap are, respectively,

$$\begin{aligned} E = & l_1 p_1 + (l_1 + l_2) p_2 + (l_1 + l_2 + l_3) p_3 + \dots \\ & + (l_1 + l_2 + l_3 + \dots + l_{k-1} + l_k) p_k \\ & + (l_1 + l_2 + l_3 + \dots + l_{k-1} + l_k + l_{k+1}) p_{k+1} \\ & + \dots + (l_1 + l_2 + l_3 + \dots + l_n) p_n \end{aligned}$$

and

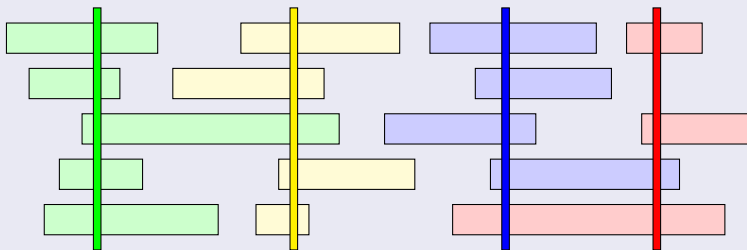
$$\begin{aligned} E' = & l_1 p_1 + (l_1 + l_2) p_2 + (l_1 + l_2 + l_3) p_3 + \dots \\ & + (l_1 + l_2 + l_3 + \dots + l_{k-1} + l_{k+1}) p_{k+1} \\ & + (l_1 + l_2 + l_3 + \dots + l_{k-1} + l_k + l_{k+1}) p_k \\ & + \dots + (l_1 + l_2 + l_3 + \dots + l_n) p_n. \end{aligned}$$

- Thus,  $E - E' = l_k p_{k+1} - l_{k+1} p_k$ , which is positive whenever  $l_k p_{k+1} > l_{k+1} p_k$ , i.e. when  $p_k/l_k < p_{k+1}/l_{k+1}$ .
- Consequently,  $E > E'$  if and only if  $p_k/l_k < p_{k+1}/l_{k+1}$ , which means that the swap decreases the expected time whenever  $p_k/l_k < p_{k+1}/l_{k+1}$ , i.e., if there is an inversion: file  $k + 1$  with a larger ratio  $p_{k+1}/l_{k+1}$  has been put after file  $k$  with a smaller ratio  $p_k/l_k$ .
- As long as the sequence is not sorted, there will be inversions of consecutive files, and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

- Note: We DO NOT use the Bubble Sort in our algorithm, it is used only in the proof that optimal solution contains no inversions.
- In the algorithm itself we compute in time  $O(n)$  all the ratios  $p_k/l_k$  and then sort all files according to these ratios in time  $O(n \log n)$ .
- Thus, the whole algorithm runs in time  $O(n \log n)$ , while Bubble Sort can sometimes run in time  $O(n^2)$ .

## Problem

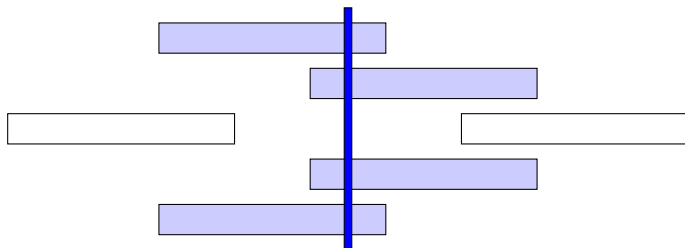
**Instance:** Let  $X$  be a set of  $n$  intervals on the real line, described by two arrays  $X_L[1..n]$  and  $X_R[1..n]$ , representing their left and right endpoints. We say that a set  $P$  of points stabs  $X$  if every interval in  $X$  contains at least one point in  $P$ .



**Task:** Describe and analyse an efficient algorithm to compute the smallest set of points that stabs  $X$ .

## Attempt 1

Is it a good idea to stab the largest possible number of intervals?



No! In the above example, this strategy needs three stabbing points rather than two.

## Hint

The interval which ends the earliest has to be stabbed somewhere.

What is the best place to stab it?

## Problem

**Instance:** A list of  $n$  powder substances, described by their weights  $w_i$  and values  $v_i$ , and a maximal weight limit  $W$  of your knapsack. You can take any fraction available of each substance (not necessarily integer).

**Task:** Select a non-negative quantity of each substance, with total weight not exceeding  $W$  and maximal total value.



## Solution

Take maximal available amount of the substance of highest value per unit weight!

### Problem

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

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

- Can we always choose the item of highest value per unit weight?
- Assume there are just three items with weights and values:

$A(10 \text{ kg}, \$60), B(20 \text{ kg}, \$100), C(30 \text{ kg}, \$120)$

and a knapsack of capacity  $W = 50 \text{ kg}$ .

- The greedy strategy would choose items  $A$  and  $B$ , while the optimal solution is to take items  $B$  and  $C$ !
- So when does the Greedy Strategy work??
- As we mentioned, unfortunately there is no easy rule ...

- 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!

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings, so that sequences of such symbols can be decoded in an unambiguous way.

- One way of doing so is to reserve bit strings of equal and sufficient length, given the number of distinct symbols to be encoded. For example, if you have 26 letters and up to 6 punctuation symbols, you could use strings of 5 bits, as  $2^5 = 32$ .
- To decode a piece of text you would partition the bit stream into groups of 5 bits and use a lookup table to decode the text.

- However this might not be the most economical way: all the symbols have codes of equal length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symbols such as 'a', 'e', 'i' or 't' have short codes while infrequent ones, such as 'q', 'x' and 'z' can have longer codes.

- However, if the codes are of variable length, then how can we partition a bitstream *uniquely* into segments each corresponding to a code?
- One way of ensuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with this property are called *prefix codes*.



- We can now formulate the problem as follows:

*Given the frequencies (probabilities of occurrences) of each symbol, design an optimal prefix code, i.e. a prefix code which minimises the expected length of an encoded text.*

- Note that this amounts to saying that the *average* number of bits per symbol in an “average” text is as small as possible.
- We now sketch the algorithm informally; please see the textbook for details and the proof of optimality.
- MATH3411 Information, Codes & Ciphers covers this and much more!

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

### 3.1 Directed graph structure

### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

1. Introduction

2. Assorted problems

3. Applications to graphs

3.1 Directed graph structure

3.2 Single source shortest paths

3.3 Minimum spanning trees

4. Puzzle

## Problem

**Instance:** There are  $n$  radio towers for broadcasting tsunami warnings. You are given the  $(x, y)$  coordinates of each tower and its radius of range. When a tower is activated, all towers within the radius of range of the tower will also activate, and those can cause other towers to activate and so on.

You need to equip some of these towers with seismic sensors so that when these sensors activate the towers where these sensors are located all towers will eventually get activated and send a tsunami warning.

**Task:** Design an algorithm which finds the fewest number of towers you must equip with seismic sensors.

## Attempt 1

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

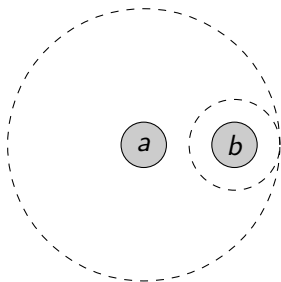
## Attempt 2

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.

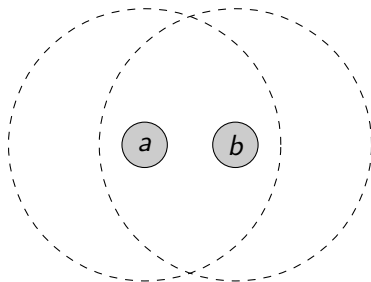
## Exercise

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

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 radius of  $a$ .



$$a \rightarrow b$$



$$a \leftrightarrow b$$

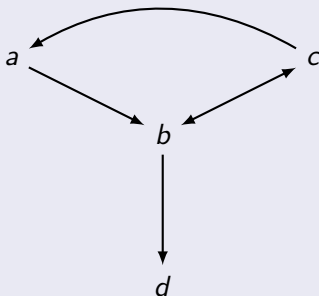
## Observation

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.
- Can we do better?

## Example



All four towers can be activated by placing just one sensor at  $a$ ,  $b$  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*.

## Definition

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

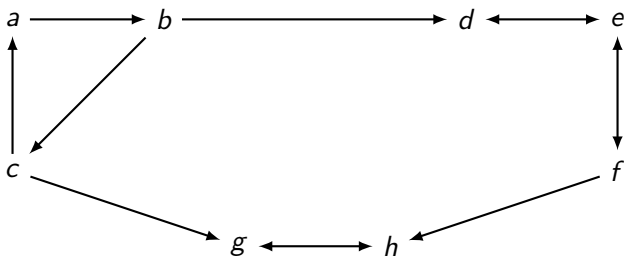
- How do we find the strongly connected component  $C_v \subseteq V$  containing  $v$ ?
- Construct another graph  $G_{rev} = (V, E_{rev})$  consisting of the same set of vertices  $V$  but with the set of edges  $E_{rev}$  obtained by reversing the direction of all edges  $E$  of  $G$ .

## Claim

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

Equivalently,  $u$  is reachable from  $v$  in both  $G$  and  $G_{rev}$ .

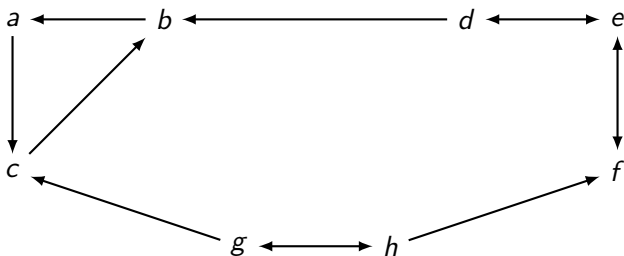
Suppose the original graph  $G = (V, E)$  is



Then the set of vertices reachable from  $e$  is

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

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



Then the set of vertices reachable from  $e$  is

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

- Combining

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

with

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

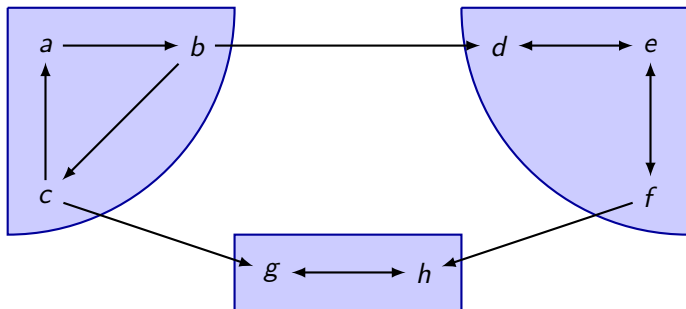
we have

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

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

- 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  $C_v = R_v \cap R'_v$ .

Therefore the decomposition of  $G$  into strongly connected components is





- 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  $O(V(V + E))$ .
- Faster algorithms exist! Kosaraju's algorithm and Tarjan's algorithm find all strongly connected components of a directed graph in *linear time*, i.e.  $O(V + E)$  (covered in COMP4128).

- 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  $G$ .

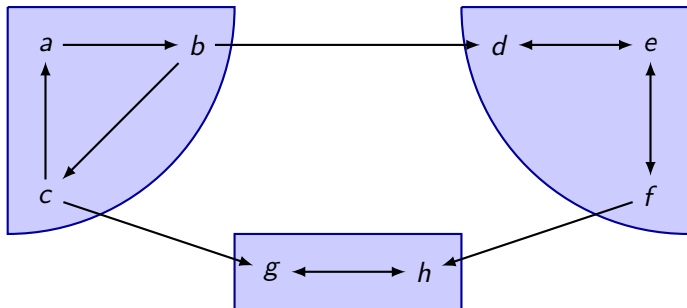
## Definition

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

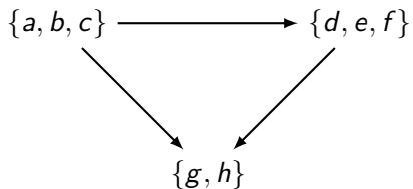
$$E^* = \{(C_{u_1}, C_{u_2}) \mid (u_1, u_2) \in E, C_{u_1} \neq C_{u_2}\}.$$

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

Recall our earlier example



The condensation graph is simply



- We begin our solution to the tsunami warning problem by finding the condensation graph.
- Now we have the set of super-towers, and we know for each super-tower 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  $\Sigma_G$  is a directed acyclic graph.

## Proof Outline

Suppose there is a cycle in  $\Sigma_G$ . Then the vertices on this cycle are not *maximal* strongly connected sets, as they can be merged into an even larger strongly connected set.

## Solution

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

## Proof

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.

## Proof (continued)

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.



## Definition

Let  $G = (V, E)$  be a directed graph, and let  $n = |V|$ . A *topological sort* of  $G$  is a linear ordering (enumeration) 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

A directed acyclic graph permits a topological sort of its vertices.

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

## Algorithm (continued)

While set  $S$  is non-empty, select a vertex  $u$  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.

This algorithm runs in  $O(V + E)$ , that is, *linear time*.

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

### 3.1 Directed graph structure

### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

## Problem

**Instance:** a directed graph  $G = (V, E)$  with *non-negative* weight  $w(e)$ , and a designated *source* vertex  $s \in V$ .

We will assume that for every  $v \in V$  there is a path from  $s$  to  $v$ .

**Task:** find the weight of the shortest path from  $s$  to  $v$  for every  $v \in V$ .

## 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.

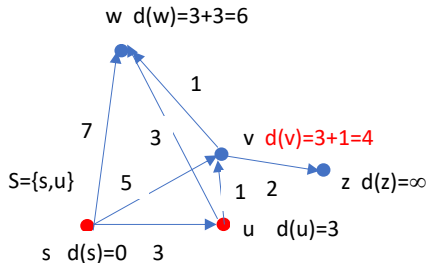
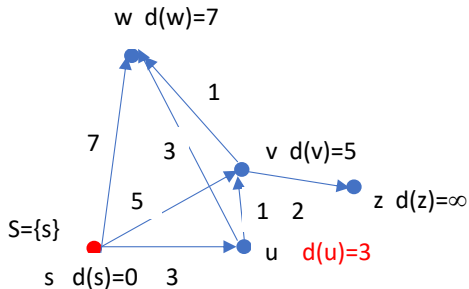
This task is accomplished by a very elegant greedy algorithm developed by Edsger Dijkstra in 1959.

## Algorithm Outline

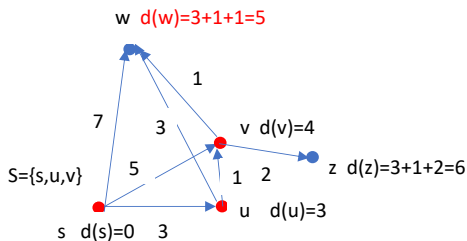
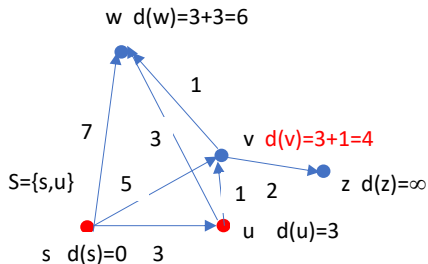
Maintain a set  $S$  of vertices for which the shortest path weight has been found, initially empty.  $S$  is represented by a boolean array.

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.







This outline still leaves much work to do.

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

First, we will prove the correctness of Dijkstra's algorithm.

## 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

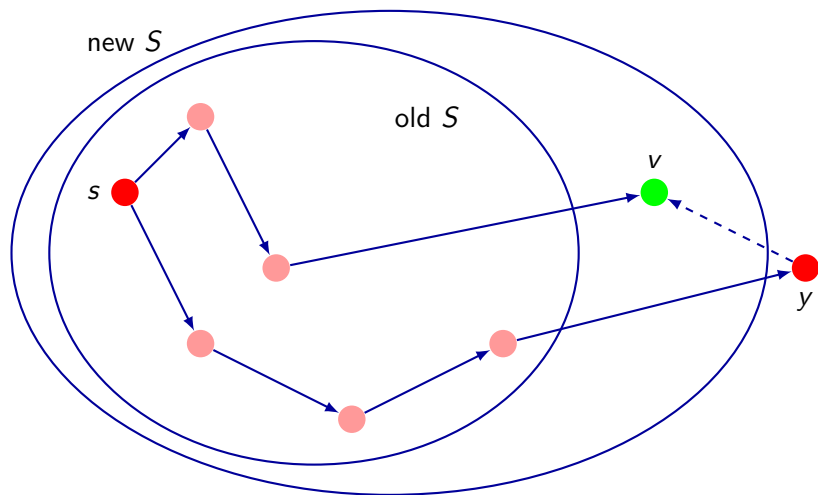
- $d_v$  is the length of the shortest path from  $s$  to  $v$  using only intermediate vertices in  $S$ . Let's call this path  $p$ .
- If this were *not* to be the shortest path from  $s$  to  $v$ , there must be some shorter path  $p'$  which first leaves  $S$  at some vertex  $y$  before later reaching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

## Proof (continued)

- 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  $d_y$ .

## Proof (continued)

- 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 the shortest path from  $s$  to  $v$ .



## 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!

## 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

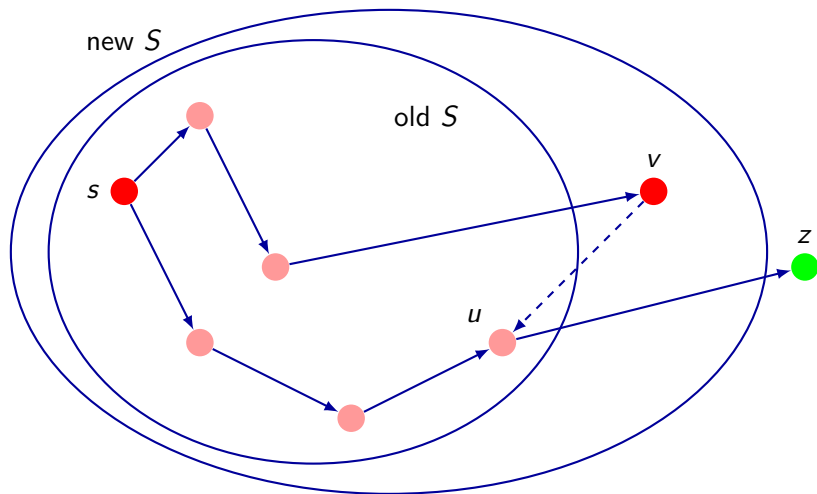
- Suppose that adding  $v$  to  $S$  allows for a new shortest path through  $S$  from  $s$  to  $z$  with penultimate vertex  $u \neq v$ .
- Such a path must include  $v$ , or else it would not be new. Thus the path is of the form

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



## Proof (continued)

- 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 from  $u$  to  $z$  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.



- 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.

Let  $n = |V|$  and  $m = |E|$ , and suppose the vertices of the graph are labelled  $1, \dots, n$ , where 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  $O(n^2 + m)$ .

- In a simple graph (no self loops or parallel edges) we have  $m \leq n(n - 1)$ , so we can simplify the time complexity expression to just  $O(n^2)$ .
- If the graph is *dense*, this is fine. But this is not guaranteed!
- Can we do better when  $m \ll n^2$ ?

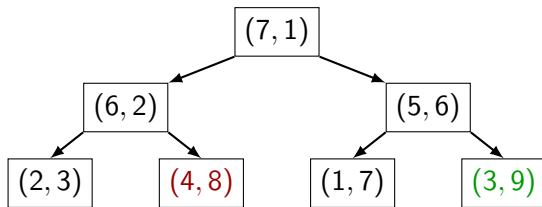
- Recall the two operations we need to support:
  - find the vertex  $v \in V \setminus S$  with smallest  $d_v$  value, and
  - for each of its outgoing edges  $(v, z)$ , update  $d_z$  if necessary.
- So far, we have done the first operation in  $O(n)$  using a linear search.
- How can we improve on 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.



- We will 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 will 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  $A[P[i]] = (i, d_i)$ .

- Changing 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  $O(1)$ , and popping the heap still takes  $O(\log n)$ .



$j$	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

## 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.

## Question

What is the time complexity of our algorithm?

## Answer

- Each of  $n$  stages requires a deletion from the heap (when a vertex is added to  $S$ ), which takes  $O(\log n)$  many steps.
- Each edge causes at most one update of a key in the heap, also taking  $O(\log n)$  many 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  $m \geq n - 1$ , so we can simplify to  $O(m \log n)$ .

## 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.

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

### 3.1 Directed graph structure

### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

## Definition

A minimum spanning tree  $T$  of a connected graph  $G$  is a subgraph of  $G$  (with the same set of vertices) which is a tree, and among all such trees it minimises the total length of all edges in  $T$ .

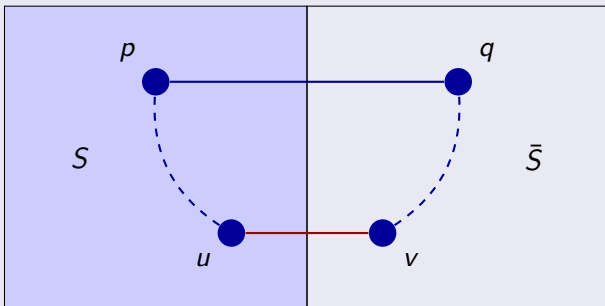
## Lemma

Let  $G$  be a connected graph with all lengths of edges  $E$  of  $G$  distinct and  $S$  a non empty proper subset of the set of all vertices  $V$  of  $G$ . Assume that  $e = (u, v)$  is an edge such that  $u \in S$  and  $v \notin S$  and is of minimal length among all the edges having this property. Then  $e$  must belong to every minimum spanning tree  $T$  of  $G$ .



## Proof

Assume that there exists a minimum spanning tree  $T$  which does not contain such an edge  $e = (u, v)$ .



## Proof (continued)

- Since  $T$  is a spanning tree, there exists 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.

- 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*.

- 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.

## Claim

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

## Proof

We consider the case when all weights are distinct.

- Consider an edge  $e = (u, v)$  added in the course of Kruskal's algorithm, and let  $F$  be the forest in its state *before* adding  $e$ .

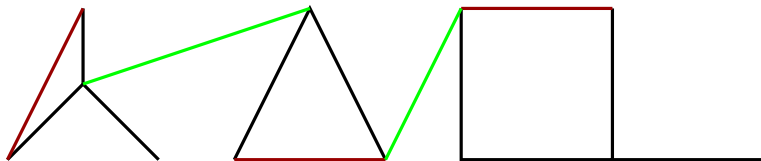
## Proof (continued)

- Let  $S$  be the set of vertices reachable from  $u$  in  $F$ . Then clearly  $u \in S$  but  $v \notin S$ .
- The original graph does not contain any edges shorter than  $e$  with one end in  $S$  and the other outside  $S$ . If such an edge existed, it would have been considered before  $e$  and included in  $F$ , but then both its endpoints would be in  $S$ , contradicting the definition.
- Consequently, edge  $e$  is the shortest edge between a vertex of  $S$  and a vertex of  $\bar{S}$  and by the previous lemma it must belong to every minimum spanning tree.

## Proof (continued)

- Thus, the set of edges produced by Kruskal's algorithm is a subset of the set of edges of every minimum spanning tree.
- But the graph produced by Kruskal's algorithm by definition has no cycles and is connected, so it is a tree.
- Therefore in the case where all edge weights are distinct, Kruskal's algorithm produces the unique minimum spanning tree.

- 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.





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

- 1 **MAKEUNIONFIND**( $S$ ), which returns a structure in which all elements (vertices) are placed into distinct singleton sets. This operation runs in time  $O(n)$  where  $n = |S|$ .
- 2 **FIND**( $a$ ), which returns the (label of the) set to which  $a$  belongs. This operation runs in time  $O(1)$ .
- 3 **UNION**( $a, b$ ), which changes the data structure by merging the sets  $A$  and  $B$  (whose labels are  $a$  and  $b$  respectively) into a single set  $A \cup B$ . The first  $k$  **UNION** operations run in total time  $O(k \log k)$ .

- Note that we do not give the run time of a single UNION operation but of a sequence of  $k$  UNION operations.
- This is called *amortized analysis*; it effectively estimates the average cost of each operation in a sequence.
- Any one UNION operation might be  $\Theta(n)$ , but the total time taken by the first  $k$  is  $O(k \log k)$ , i.e. each takes 'on average'  $O(\log k)$ .
- This is different to average case analysis, because it's a statement about an *aggregate*, rather than a *probability*.

- We will label each set by one of its elements, called the *representative* of the set.
- The simplest implementation of the Union-Find data structure consists of:
  - an array  $A$ , where  $A[i] = j$  means that  $i$  belongs to the set with representative  $j$ ;
  - an array  $B$ , where  $B[i]$  contains the number of elements in the set with representative  $i$ ;
  - an array  $L$ , where  $L[i]$  contains pointers to the head and tail of a linked list containing the elements of the set with representative  $i$ .

### Note

If  $i$  is not the representative of any set, then  $B[i]$  is zero and the list  $L[i]$  is empty.

### Note

The list array  $L$  allows us to iterate through the members of one of the disjoint sets, which is used in the UNION operation.

Given two sets  $I$  and  $J$  with representatives  $i$  and  $j$ ,  $\text{UNION}(i, j)$  is defined as follows:

- assume  $B[i] \geq B[j]$  (i.e.  $|I| \geq |J|$ ); otherwise perform  $\text{UNION}(j, i)$  instead;
- for each  $m \in J$ , use linked list  $L[j]$  to update  $A[m]$  from  $j$  to  $i$ ;
- update  $B[i]$  to  $B[i] + B[j]$  and  $B[j]$  to zero;
- append the list  $L[j]$  to the list  $L[i]$  and replace  $L[j]$  with an empty list.

### Observation

The new value of  $B[i]$  is at least twice the old value of  $B[j]$ .

### Observation

Suppose  $m$  is an element of the smaller set  $J$ , so its label  $A[m]$  changed from  $j$  to  $i$ .

Then the observation above tells us that  $B[A[m]]$  (formerly the old  $B[j]$ , now the new  $B[i]$ ) at least doubled.

What's the significance of  $B[A[m]]$ ? It's the number of elements in the set containing  $m$ .

- The first  $k$  UNION operations can touch at most  $2k$  elements of  $S$  (with equality if they each merge a different pair of singleton sets).
- Thus, the set containing an element  $m$  after the first  $k$  UNION operations must have at most  $2k$  elements.
- Since every UNION operation which changes  $A[m]$  at least doubles  $B[A[m]]$ , we deduce that  $A[m]$  has changed at most  $\log 2k$  times.
- Thus, since at most  $2k$  elements have their label changed at all, we can conclude that the first  $k$  UNION operations will cause at most  $2k \log 2k$  label changes in  $A$ .

- Each UNION operation requires only constant time to update the size array  $B$  and the list array  $L$ .
- Thus, the first  $k$  UNION operations take  $O(k \log k)$  time in total.
- This Union-Find data structure is good enough to get the sharpest possible bound on the run time of Kruskal's algorithm.
- See the textbook for a Union-Find data structure based on pointers and path compression, which further reduces the amortised complexity of the UNION operation at the cost of increasing the complexity of the FIND operation from  $O(1)$  to  $O(\log n)$ .



- 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.

- For each edge  $e = (u, v)$  on the sorted list of edges, we use two **FIND** operations to determine whether vertices  $u$  and  $v$  belong to the same component.
- If they do not belong to the same component, i.e., if  $\text{FIND}(u) = i$  and  $\text{FIND}(v) = j$  where  $i \neq j$ , we add edge  $e$  to the spanning tree being constructed and perform **UNION**( $i, j$ ) to merge the connected components containing  $u$  and  $v$ .
- If instead  $\text{FIND}(u) = \text{FIND}(v)$ , there is already a path between  $u$  and  $v$ , so adding this edge would create a cycle. Therefore, we simply discard the edge.

- We perform  $2m$  FIND operations, each costing  $O(1)$ .
- We also perform  $n - 1$  UNION operations, which in total cost  $O(n \log n)$ .
- The overall time complexity is therefore  $O(m \log n + m + n \log n)$ .
- The first term (from sorting) dominates, so we can simplify the time complexity to  $O(m \log n)$ .

## Problem

**Instance:** A complete graph  $G$  with weighted edges representing distances between the two vertices.

**Task:** Partition the vertices of  $G$  into  $k$  disjoint subsets so that the minimal distance between two points belonging to different sets of the partition is as large as possible. Thus, we want a partition into  $k$  disjoint sets which are as far apart as possible.

## Solution

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.

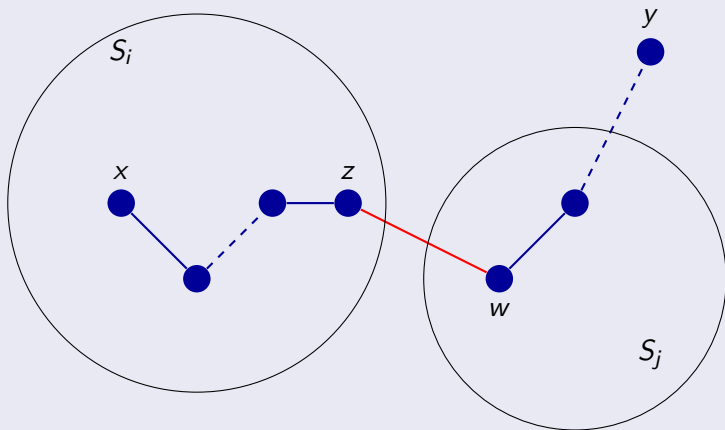
## Proof of optimality

- Let  $\delta$  be the distance associated with the first edge of the minimal spanning tree which was not added to our  $k$  connected components.
- It is clear that  $\delta$  is the minimal distance between two vertices belonging to different connected components.
- All the edges included in the connected components produced by our algorithm are of length at most  $\delta$ .

## Proof of optimality (continued)

- Consider any partition  $\mathcal{S}$  into  $k$  subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices  $x$  and  $y$  such that  $x \in S_i$  and  $y \notin S_i$  for some  $S_i \in \mathcal{S}$ .

## Proof of optimality (continued)





## Proof of optimality (continued)

- Since  $x$  and  $y$  belong to the same connected component, there is a path in that component connecting  $x$  and  $y$ .
- Let  $z$  and  $w$  be two consecutive vertices on that path such that  $z$  belongs to  $S_i$  and  $w \notin S_i$ .
- Thus,  $w \in S_j$  for some  $j \neq i$ .

## Proof of optimality (continued)

- Since  $(z, w)$  was an edge chosen by our proposed algorithm, we know that  $d(z, w) \leq \delta$ .
- It follows that the distance between these two clusters  $S_i, S_j \in \mathcal{S}$  is at most  $\delta$ .
- Thus, such a partition cannot be a better clustering than the one produced by our algorithm.

- 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  $O(n^2 \log n)$ .
- Running the (partial) Kruskal algorithm requires  $O(m \log n)$  steps, making use of the UNION-FIND data structure. Since  $m = \Theta(n^2)$ , this step also takes  $O(n^2 \log n)$ .
- So the algorithm has time complexity  $O(n^2 \log n)$  in total.

## 1. Introduction

## 2. Assorted problems

## 3. Applications to graphs

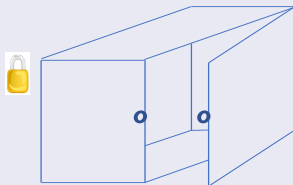
### 3.1 Directed graph structure

### 3.2 Single source shortest paths

### 3.3 Minimum spanning trees

## 4. Puzzle

## Problem



Bob is visiting Elbonia and wishes to send his teddy bear to Alice, who is staying at a different hotel. Both Bob and Alice have boxes like the one illustrated above, as well as padlocks which can be used to lock the boxes.

### Problem (continued)

However, there is a problem. The Elbonian postal service mandates that when a nonempty box is sent, it must be locked. Also, they do not allow keys to be sent, so the key must remain with the sender. Finally, you can send padlocks only if they are locked. How can Bob safely send his teddy bear to Alice?

### Hint

The way in which the boxes are locked (via a padlock) is important. It is also crucial that *both* Bob and Alice have padlocks and boxes. They can also communicate over the phone to agree on the strategy.

There are two possible solutions; one can be called the “AND” solution, the other can be called the “OR” solution. The “AND” solution requires 4 mail one way services while the “OR” solution requires only 2.



**That's All, Folks!!**