Module 3: Greedy Algorithms

Raveen de Silva (K17 202)
Course Admins: cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
UNSW Sydney

Term 1, 2024

Module Learning Outcomes

- Explain how greedy techniques are used to develop algorithms
- Solve problems by creatively applying greedy techniques
- Communicate algorithmic ideas at different abstraction levels
- Evaluate the efficiency of algorithms and justify their correctness
- Apply the LaTeX typesetting system to produce high-quality technical documents

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

Example problem: Frozen Yoghurt I

Problem

There are n flavours of frozen yoghurt at your local shop. The ith flavour is dispensed from a machine of capacity c_i litres and contributes d_i/c_i deliciousness per litre, i.e. the entire machine's worth contributes d_i.

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

Fractional Knapsack

Problem

Instance: A list of n items described by their weights w_i and values v_i, and a maximal 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 maximal total value.

Fractional Knapsack

Solution

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

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

Example problem: Frozen Yoghurt II

Problem

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 ith 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.

0-1 Knapsack

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.

0-1 Knapsack

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

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

  and a knapsack of capacity $W = 50\,\mathrm{kg}$.
- 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 . . .

Question

What is a greedy algorithm?

Answer

A greedy algorithm is one 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.

Greedy Algorithms

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.

Question

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

Answer

Yes, but we won't use it.

The study of *matroids* is 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?

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

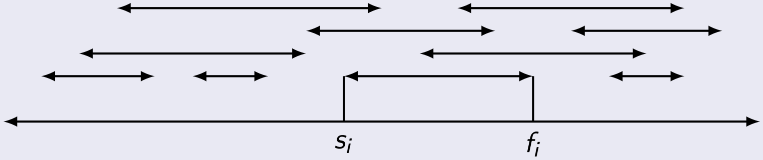
4. Applications to graphs
4.1 Directed graph structure
4.2 Single source shortest paths
4.3 Minimum spanning trees

5. Puzzle

Activity Selection

Problem

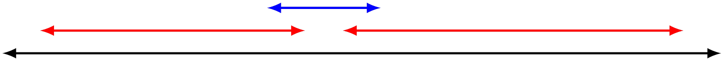
Instance: A list of n activities, with starting times s_i and finishing times f_i. No two activities can take place simultaneously.



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

Attempt 1

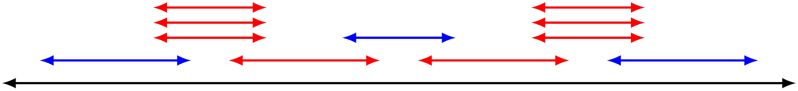
Always choose the shortest activity which does not conflict with the previously chosen activities, then remove the conflicting activities and repeat.



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

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 . . .

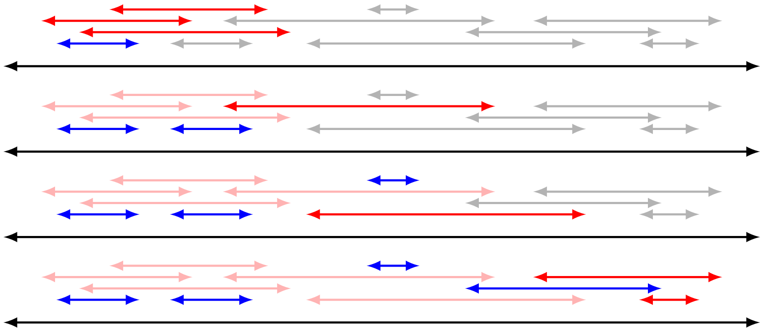


As appealing this idea is, the above figure shows this again does not work!

Activity Selection

Solution

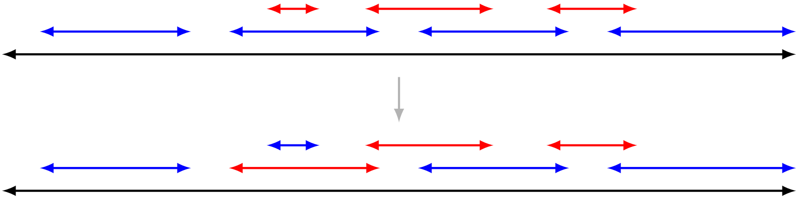
Among those activities which do not conflict with the previously chosen activities, always choose the activity with the earliest end time (breaking ties arbitrarily).



Activity Selection

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.

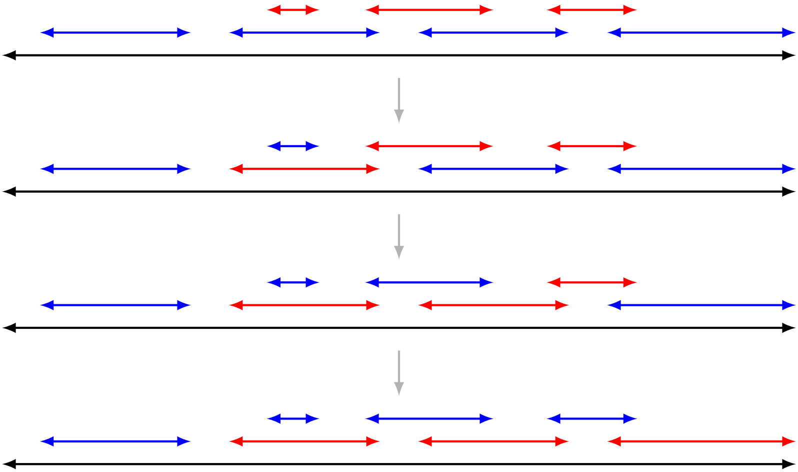
- Find the first place where the chosen activity violates the greedy choice.
- What if we replace that activity with the greedy choice?



Activity Selection: proof outline

- 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 without worsening the objective (number of activities selected).

Activity Selection



Activity Selection: Proof

- Suppose the greedy selects activities $G = \{g_1, \ldots, g_r\}$, and consider some alternative selection $A = \{a_1, \ldots, a_s\}$, each in ascending order of ending time.
  - Our aim is to prove that no other valid selection can have more activities, i.e. $s \leq r$ **regardless of how A was chosen**.
  - We don't get to choose how this alternative selection was made, which is why we haven't provided an algorithm for it.
- Suppose $g_1 = a_1, g_2 = a_2, \ldots, g_{k-1} = a_{k-1}$, but $g_k \neq a_k$.
- Then we will be comparing

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

with

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

Activity Selection: Proof

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

Is A' valid?

- There are no conflicts among $\{g_1, \ldots, g_{k-1}, a_{k+1}, \ldots, a_s\}$; these were all in the valid selection A.
- g_k doesn't conflict with $\{g_1, \ldots, g_{k-1}\}$; these are all in the valid selection G.
- g_k finishes no later than a_k, which in turn finishes before any of $a_{k+1}, \ldots, a_s$ start.

Therefore A' is a valid selection, i.e. it has no conflicts.

Activity Selection: Proof

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

How big is A'?

- g_k isn't a duplicate of any of $\{g_1, \ldots, g_{k-1}\}$ as G has no duplicates.
- g_k isn't a duplicate of any of $\{a_{k+1}, \ldots, a_s\}$ as it finishes before they start.

Therefore A' contains s intervals, just as A did.

Activity Selection: Proof

- We started with an alternative selection A, and the first time it disagreed with G we showed that the modified alternative A' without this disagreement was just as good.[1]
- We could now consider the first point of disagreement between G and A', and resolve it in a further modified alternative A'' that is again just as good.
- Continuing in this way, we can resolve all disagreements between the greedy and the alternative until one of them runs out of activities.
- The greedy can't run out first; if a_{r+1} doesn't conflict with $g_1, \ldots, g_r$, then the greedy wouldn't end after picking g_r.
- Therefore $s \leq r$, i.e. no alternative selection can take more activities than our greedy algorithm does.

[1] In general, we need A' to be no worse than A, i.e. equal or better in terms of the objective.

Activity Selection

- 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?

Activity Selection

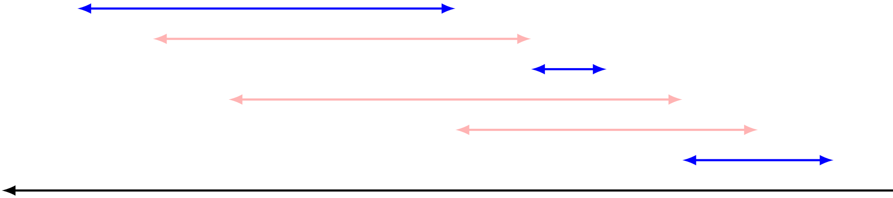
- Suppose we are up to activity i, starting at s_i and finishing at f_i, with all earlier finishing activities already processed.

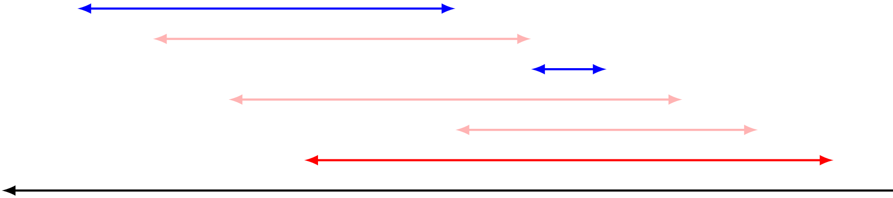
- If all previously chosen activities finished before s_i, activity i can be chosen without a conflict.

- Otherwise, there will be a clash, so we discard activity i.

- We would prefer not to go through all previously chosen activities each time.

Activity Selection





Activity Selection

- 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.

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. 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.

Activity Selection

Question

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

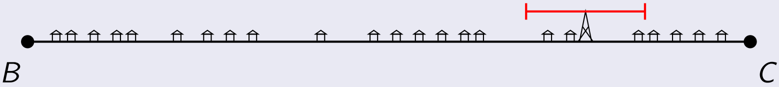
Solution

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 true anymore.

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

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 minimal 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 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.

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

- For variety, we will present an alternative method of proof called *greedy stays ahead*.

- Where the exchange argument is akin to proof by contradiction, greedy stays ahead is more similar to proof by induction.

- The key claim will be: for all k, the kth tower in the greedy solution is at least as far east as the kth tower in any other placement.

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

Cell Towers: Proof

- The base case is $k = 1$.
  - The greedy places its first tower five kilometres east of the first house.
  - If an alternative placement had its first tower further east, it would not cover this first house, so the claim is true for $k = 1$.
- Suppose the claim is true for some $k - 1$, and consider the kth tower.
  - The greedy placed towers at g_{k-1} and g_k, and there is a house h such that
  $$g_{k-1} + 5 < h = g_k - 5.$$
  - The alternative placed towers at a_{k-1} and a_k. The induction hypothesis tells us that $a_{k-1} \leq g_{k-1}$, so if $a_k > g_k$ then house h will not be covered by either of the towers either side of it. Therefore $a_k \leq g_k$ as required.

Cell Towers: Proof

- Now suppose the 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 $h > g_{r-1} + 5$.
- From the induction above, 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 can't be covered in the first $r - 1$ towers in *any* placement of towers.
- So at least r towers are needed, and therefore the greedy is optimal.

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

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.

Tape Storage

- Intuitively, we might expect the ideal arrangement to put the shortest files first, and the longest files at the end.
- How can we convince a reader (or ourselves) of this? Start by quantifying the retrieval time.
- If the files are stored in order $l_1, l_2, \ldots, l_n$, then the expected time is

$$\frac{1}{n} l_1 + \frac{1}{n}(l_1 + l_2) + \frac{1}{n}(l_1 + l_2 + l_3) + \ldots + \frac{1}{n}(l_1 + l_2 + l_3 + \ldots + l_n).$$

- Grouping like terms gives

$$\frac{1}{n}\left(n l_1 + (n-1) l_2 + (n-2) l_3 + \ldots + 2 l_{n-1} + l_n\right).$$

- We'll ignore the common factor $\frac{1}{n}$, as we have no way of influencing it.

Tape Storage

Conjecture

The ‘scaled’ retrieval time

$$nl_1 + (n-1)l_2 + (n-2)l_3 + \ldots + 2l_{n-1} + l_n,$$

is minimised if $l_1 \leq l_2 \leq \ldots \leq l_n$.

Tape Storage

- We use perhaps the most common argument for optimal ordering, a form of proof by contradiction.

- Suppose the files are *not* in ascending order by length. Then there must be two consecutive files which are out of order, i.e. $l_i > l_{i+1}$.

- Swapping them *reduces* the scaled retrieval time from

$$\ldots + (n-i+1)l_i + (n-i)l_{i+1} + \ldots$$

to

$$\ldots + (n-i+1)l_{i+1} + (n-i)l_i + \ldots.$$

Tape Storage

- 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, à la bubble sort.
- Note that this doesn't mean the files should be sorted 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

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.

Tape Storage II

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

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

- There are two obvious heuristics:
  - short files should again be at the start, but now
  - frequently accessed files should also be at the start.
- We can't fully rank the files in terms of both metrics at the same time!
  - Who is the tallest oldest person on a train?
- Compromise?

Tape Storage II

- 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,

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

and

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

Tape Storage II

- 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 $p_k / l_k < p_{k+1} / l_{k+1}$.

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

Minimising Job Lateness

Problem

Instance: A list of n jobs, with durations and 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 $\text{finish}_i > \text{due}_i$ then we say that it has incurred lateness $\text{late}_i = \text{finish}_i - \text{due}_i$.

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

Minimising Job Lateness

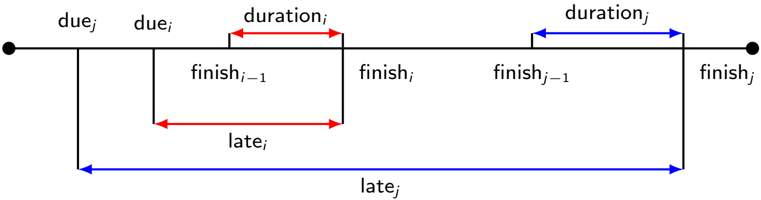
- Again, our intuition suggests two heuristics:
  - jobs of short duration should be scheduled first, and
  - jobs due earlier should be scheduled first.
- Perhaps surprisingly, there is no compromise to be made here!

Claim

We should ignore job durations and schedule jobs in ascending order of deadlines.

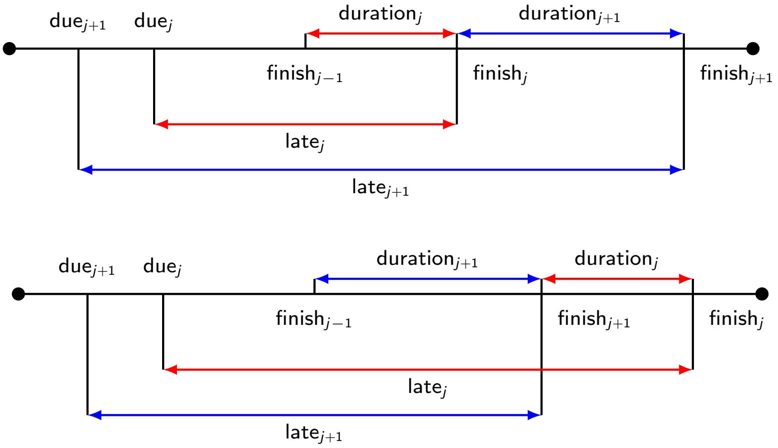
Minimising Job Lateness

- We'l say that jobs i and j form an inversion if job i is scheduled before job j but due after it (i.e. $\text{due}_j < \text{due}_i$).



- As usual, any sequence that isn't sorted has an adjacent inversion.

Minimising Job Lateness



Minimising Job Lateness

- Observe that swapping adjacent inverted jobs reduces the larger lateness!

- The largest lateness overall might not have changed; there might be some other job that's more late than either of these.

- All that matters is that it can't have got worse (i.e. increased) as a result of this swap.

- Again, it would have been much harder to directly analyse *non-adjacent* inversions. Swapping non-adjacent jobs would also affect the finishing times and hence latenesses of every job in between them.

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

Tsunami Warning

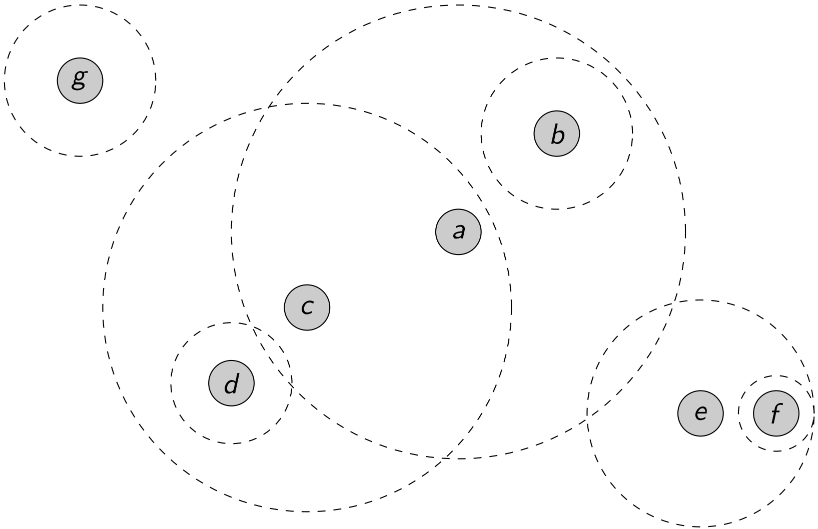
Problem

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.

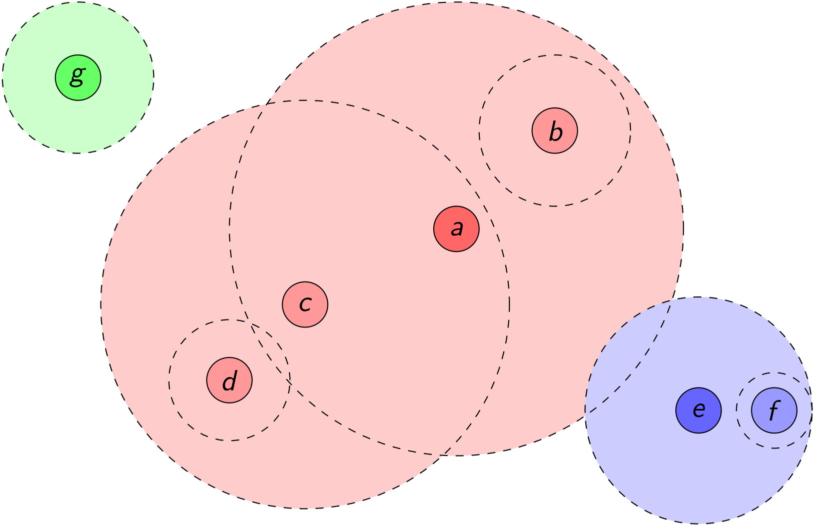
Tsunami Warning



Tsunami Warning

- Activating a causes the activation of b and c, and therefore d is activated also.
- Activating e causes the activation of f.
- g must be activated separately.
- Therefore a minimum of three sensors are required.
- Note that we could have placed the first sensor at c instead of a.

Tsunami Warning



Tsunami Warning

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

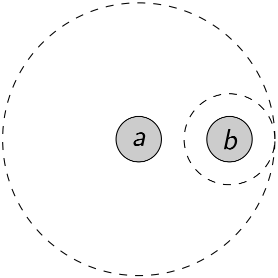
Find the unactivated tower with the largest number of towers within its range (breaking ties arbitrarily), and place a sensor at this tower. Find and remove all towers activated as a result. Repeat.

Exercise

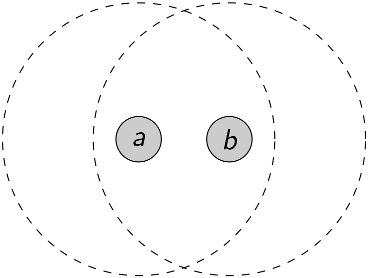
Give examples which show that neither of these algorithms solve the problem correctly.

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 radius of a.



$a \rightarrow b$



$a \leftrightarrow b$

Tsunami Warning

Observation

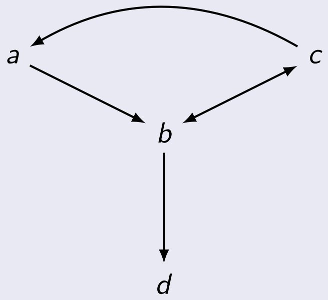
Suppose that activating tower *a* causes tower *b* to also be activated, and vice versa. Then we never want to place sensors at both towers; indeed, placing a sensor at *a is equivalent to* placing a sensor at *b*.

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

- Cycles also have this property.
- Can we do better?

Tsunami Warning

Example



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

Tsunami Warning

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

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.

Strongly Connected Components

- 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.

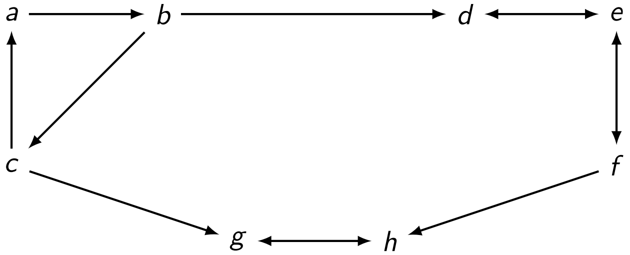
Claim

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}.

Strongly Connected Components

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

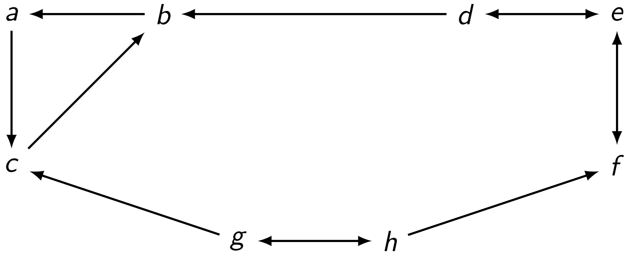


Then the set of vertices reachable from e is

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

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 $C_a = \{a, b, c\}$ and $C_g = \{g, h\}$.

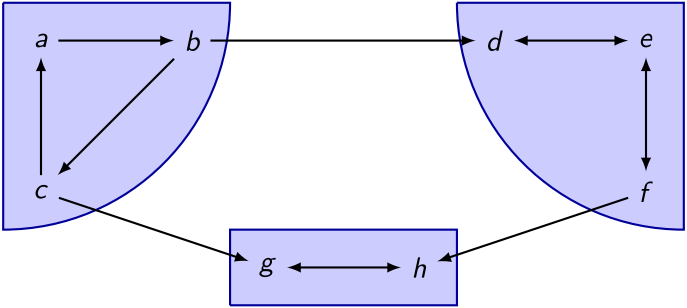
Strongly Connected Components

- 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



Strongly Connected Components

- 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 $O(V(V+E))$.

- This is brute force; can we do better?

- Could one DFS suffice?

Tarjan's Algorithm

- Do a regular DFS on the graph, but with an explicit stack.
- When an item is pushed onto the stack, mark it as "in-stack", and unmark it as such when it is popped.
- If we want to push a vertex that is already "in-stack", then we've found a strongly connected component.
- Each item on the stack after this vertex can be reached from it, and can also reach that vertex.
- Simply pop everything off the stack including that vertex, and combine it into an SCC.

Strongly Connected Components

- Kosaraju's algorithm (presented in CLRS §22.5) is an alternative.
    - It involves two uses of DFS.
    - More complicated to reason about, easier to implement in code.
- Both of these algorithms run in *linear time*, i.e. $O(V+E)$.
- 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 a problem.

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

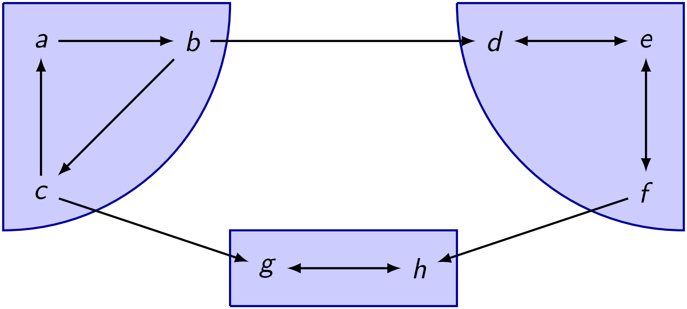
Define the *condensation graph* $\Sigma_G = (C_G, E^*)$, where

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

The vertices of 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.

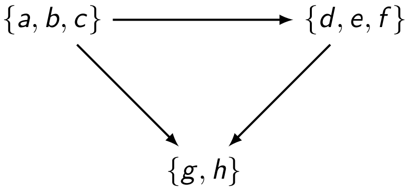
The Condensation Graph

Recall our earlier example



The Condensation Graph

The condensation graph is simply



The Condensation Graph

- 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.

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* strongly connected sets, as they can be merged into an even larger strongly connected set.

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

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.

Topological Sorting

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 $\sigma : V \to \{1, \ldots, n\}$ 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.

Topological Sorting

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.

Topological Sorting

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.

Topological Sorting

- 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 start with a topological sort 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

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

Single Source Shortest Paths

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 $v \in V$.

Single Source Shortest Paths

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.

- 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?

Dijkstra's Algorithm: Correctness

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

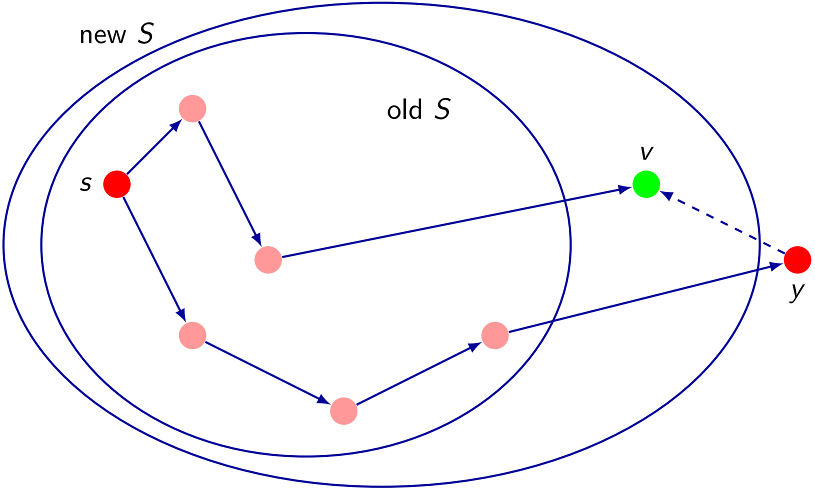
Claim

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.

Dijkstra's Algorithm: Correctness



Dijkstra's Algorithm: Correctness

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.

Dijkstra's Algorithm: Correctness

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.

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.

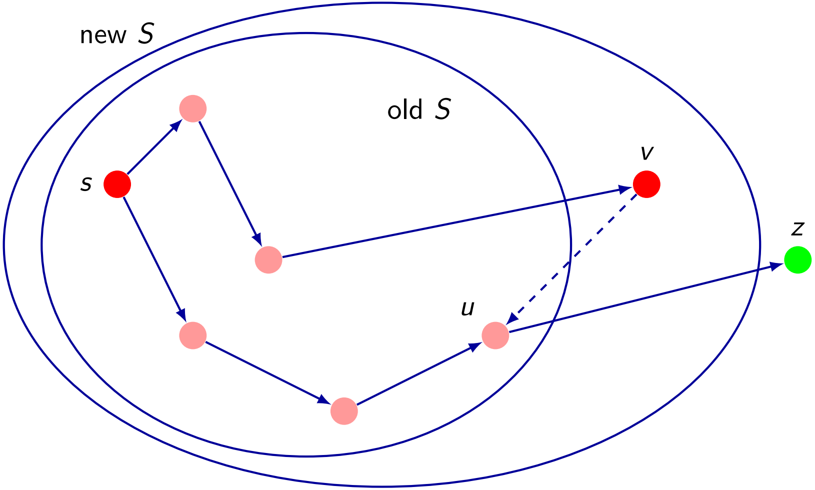
Proof

- 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

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

Dijkstra’s Algorithm: Updates



Dijkstra's Algorithm: Updates

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.

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 $m = |E|$, and suppose the vertices of the graph are labelled $1, \ldots, n$, 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.

Dijkstra's Algorithm: Array

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 $O(n^2 + m)$.

Dijkstra's Algorithm: Array

- 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 $m \ll n^2$?

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?

Dijkstra's Algorithm: Data Structures

- 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 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.

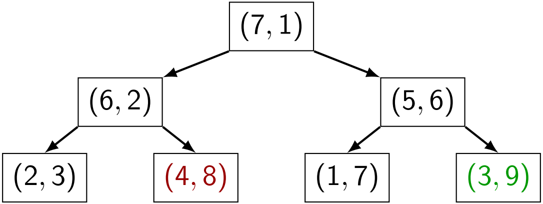
- We will also maintain another array $P[1..n]$ which stores the *position* of elements in the heap.

- Whenever $A[j]$ refers to vertex i, we record $P[i] = j$, so that we can look up vertex i using the property $A[P[i]] = (i, d_i)$.

Augmented Heaps

- 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 $O(\log n)$.

Augmented Heaps



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

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

- 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 $O(m\log n)$.

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 unless you also detail the Fibonacci heap construction and operations, and prove the improved time complexity. This will not be the intended solution for any assessable problem in our course.

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

Minimum Spanning Trees

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

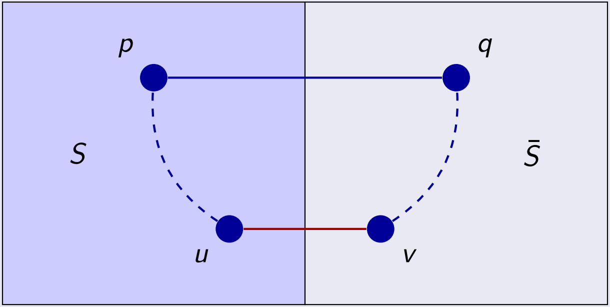
Let G be a connected graph with all lengths of edges E of G distinct and S a non empty proper subset of the set of all vertices V of G.

Let $e = (u, v)$ be a shortest edge between S and $\bar{S}$ (WLOG $u \in S, v \notin S$). Then e must belong to every minimum spanning tree T of G.

Minimum Spanning Trees

Proof

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



Minimum Spanning Trees

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.

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

- 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.

Kruskal's Algorithm

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.

Kruskal's Algorithm

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.

Kruskal's Algorithm

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.

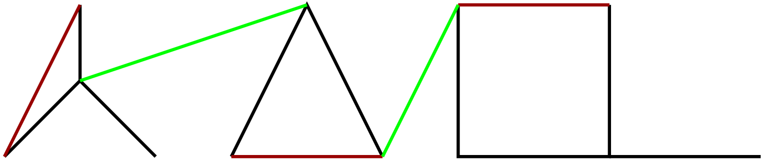
- But the graph produced by Kruskal's algorithm by definition has no cycles and is connected, so it is a tree.

- Therefore in the case where all edge weights are distinct, Kruskal's algorithm produces the unique minimum spanning tree.

Efficient Implementation of Kruskal's Algorithm

- 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.



Union-Find

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. INITIALISE(S), which returns a structure in which all items (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 $O(k \log k)$.

Union-Find

- 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*.

Union-Find

- We will label each set by one of its items.
- The simplest implementation of the Union-Find data structure consists of three arrays:
    - set: set$[i] = j$ means that i belongs to set j;
    - size: size$[j]$ is the number of items in set j;
    - elements: elements$[j]$ is a linked list of the items in set j.

Union-Find

Note

If set[i] $\neq i$, then item i has already been merged into some other set, so size[i] is zero and the list elements[i] is empty.

Note

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

Union-Find

Given items a and b belonging to sets A and B respectively, $\textsc{Union}(a, b)$ is defined as follows:

- replace a and b with set[a] and set[b] respectively;
- assume size[a] $\geq$ size[b] (i.e. $|A| \geq |B|$); otherwise perform $\textsc{Union}(b, a)$ instead;
- for each m in elements[B], update set[m] from b to a;
- update size[a] to size[a] $+$ size[b] and size[b] to zero;
- append the list elements[b] to the list elements[a] and replace elements[b] with an empty list.

Observation

The new value of size[a] is at least twice the old value of size[b].

Observation

Suppose m is an item in the smaller set B, so set[m] changed from b to a.

Then size[set[m]] changed from the old value of size[b] to the new value size[a], so it at least doubled.

How to interpret this? m is now in a set at least twice as big as it was before the UNION operation.

Union-Find

- The first k Union operations can touch at most $2k$ items of S (with equality if they each merge a different pair of singleton sets).

- Thus, the set containing an item m after the first k Union operations must have at most $2k$ items.

- Since every Union operation which changes set[m] at least doubles size[set[m]], we deduce that size[m] has changed at most $\log 2k$ times.

- Thus, since at most $2k$ items move sets at all, we can conclude that the first k Union operations will cause at most $2k \log 2k$ updates in the set array.

Union-Find

- Each Union operation requires only constant time to update the arrays size and elements.

- 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 $O(\log n)$.

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 $\textsc{Find}$ 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 $\textsc{Find}(u) = i$ and $\textsc{Find}(v) = j$ where $i \neq j$, we add edge e to the spanning tree being constructed and perform $\textsc{Union}(i, j)$ to merge the connected components containing u and v.

- If instead $\textsc{Find}(u) = \textsc{Find}(v)$, there is already a path between u and v, so adding this edge would create a cycle. Therefore, we simply discard the edge.

Efficient Implementation of Kruskal's Algorithm

- 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 $O(m \log n)$.

k-clustering of maximum spacing

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.

k-clustering of maximum spacing

Solution

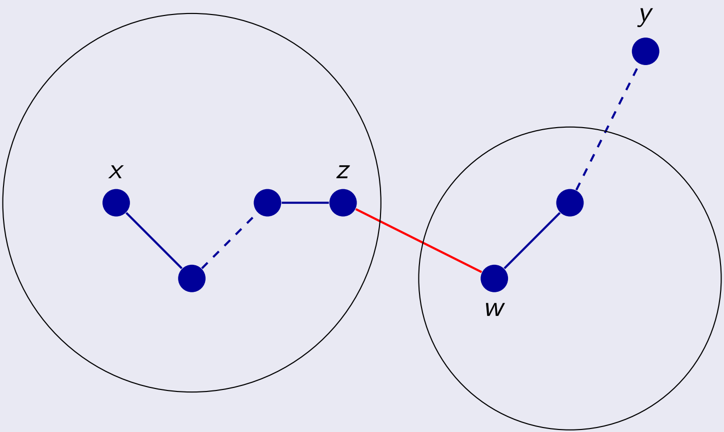
Sort the edges in increasing order and start performing the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k connected components, rather than a single spanning tree.

k-clustering of maximum spacing

Proof of optimality

- 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.
- There must be some pair of vertices x and y that are clustered together in our algorithm but are separated in the other clustering.

Proof of optimality (continued)



k-clustering of maximum spacing

Proof of optimality (continued)

- Since x and y belong to the same tree of *our* forest, there is a path 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.

- But in the other clustering (indicated by the circles in the diagram), y is not in the same cluster as x, so this path $x \to y$ 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.

Table of Contents

1. Motivation

2. Optimal Selection

3. Optimal Ordering

4. Applications to graphs

4.1 Directed graph structure

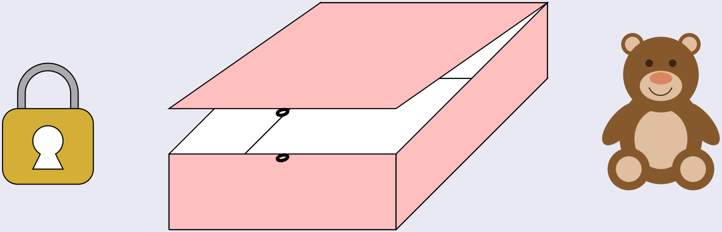
4.2 Single source shortest paths

4.3 Minimum spanning trees

5. Puzzle

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.

Puzzle

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?

Puzzle

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 three mailings while the “OR” solution requires only two.

That's All, Folks!!