Contents

IA Algorithms

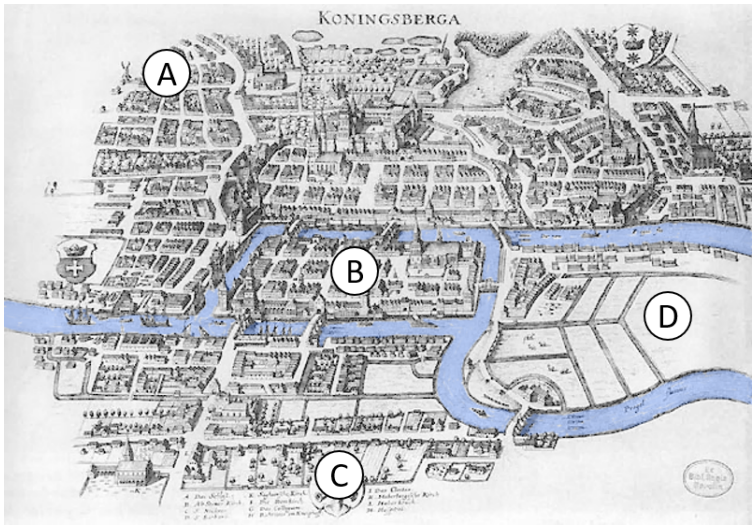
Damon Wischik, Computer Laboratory, Cambridge University. Lent Term 2018

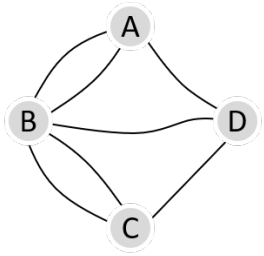
5. Graphs

5.1. Notation and representation

A great many algorithmic questions are about entities and the connections between them. Graphs are how we describe them. A graph is a set of *vertices* (or nodes, or locations) and *edges* (or connections, or links) between them.

Example. Leonard Euler in Königsberg, 1736, posed the question "Can I go for a stroll around the city on a route that crosses earch bridge exactly once?" He proved the answer was 'No'. His innovation was to turn this into a precise mathematical question about a simple discrete object—a graph.

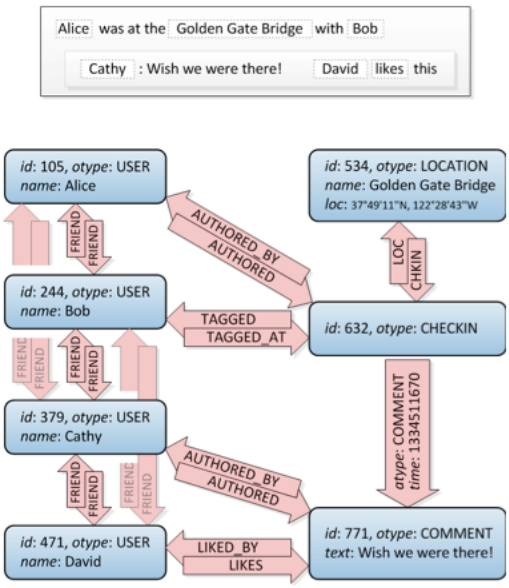





```
g = {'A': ['B', 'B', 'D'],
     'B': ['A', 'A', 'C', 'C', 'D'],
     'C': ['B', 'B', 'D'],
     'D': ['A', 'B', 'C']}
```

Example. Facebook's underlying data structure is a graph. Vertices are used to represent users, locations, comments, check-ins, etc. From the Facebook documentation,

It's up to the programmer to decide what counts as a vertex and what counts as an edge. Why do you think Facebook made CHECKIN a type of vertex, rather than an edge from a USER to a LOCATION?



Example. OpenStreetMap represents its map as XML, with nodes and ways. In some parts of the city, this data is very fine-grained. The more vertices and edges there are, the more space it takes to store the data, and the slower the algorithms run. Later in this course we

will discuss geometric algorithms which could be used to simplify the graph while keeping its basic shape.

```
<osm version="0.6" generator="Overpass API">
  <node id="687022827" user="François Guerraz"
        lat="52.2082725" lon="0.1379459" />
  <node id="687022823" user="bigalxyz123"
        lat="52.2080972" lon="0.1377715" />
  <node id="687022775" user="bigalxyz123"
        lat="52.2080032" lon="0.1376761" >
    <tag k="direction" v="clockwise"/>
    <tag k="highway" v="mini_roundabout"/>
  </node>
  <way id="3030266"  user="urViator">
    <nd ref="687022827"/>
    <nd ref="687022823"/>
    <nd ref="687022775"/>
    <tag k="cycleway" v="lane"/>
    <tag k="highway" v="primary"/>
    <tag k="name" v="East Road"/>
    <tag k="oneway" v="yes"/>
  </way>
 ...
</osm>
```

DEFINITIONS

Some notation for describing graphs:

- Denote a graph by $g = (V, E)$, where V is the set of vertices and E is the set of edges.
- A graph may be *directed* or *undirected*. For a directed graph, $v_1 \to v_2$ denotes an edge from v_1 to v_2.
- For an undirected graph, $v_1 - v_2$ denotes an edge between v_1 and v_2, the same edge as $v_2 - v_1$.
- In this course we won't allow multiple edges between a pair of nodes (such as Euler used in his graph of Königsberg bridges).
- A *path* in a directed graph is a sequence of vertices connected by edges,

$$v_1 \to v_2 \to \cdots \to v_k.$$

Paths are allowed to visit the same vertex more than once

- A *path* in an undirected graph is a sequence of vertices connected by edges,

$$v_1 - v_2 - \cdots - v_k.$$

- A *cycle* is a path from a vertex back to itself, i.e. a path where $v_1 = v_k$.

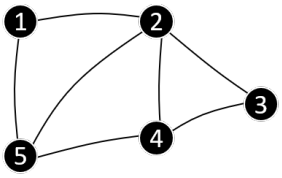
There are some special types of graph that we'll look at in more detail later.

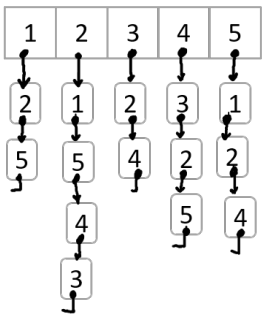
- A *directed acyclic graph* or DAG is a directed graph without any cycles. They're used all over computer science. We'll study some properties of DAGs in Section 5.10.
- An undirected graph is *connected* if for every pair of vertices there is a path between them. A *forest* is an undirected acyclic graph. A *tree* is a connected forest. We'll study algorithms for finding trees and forests in Sections 5.8–5.9.

It sounds perverse to define a tree to be a type of forest! But you need to get used to reasoning about algorithms directly from definitions, rather than from your hunches and instinct; and a deliberately perverse definition can help remind you of this.

REPRESENTATION

Here are two standard ways to store graphs in computer code: as an array of *adjacency lists*, or as an *adjacency matrix*. The former takes space $O(V + E)$ and the latter takes space $O(V^2)$, so your choice should depend on the *density* of the graph, density $= E/V^2$. (Note: V and E are sets, so we should really write $O(|V| + |E|)$ etc., but it's conventional to drop the $|\cdot|$.)





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

5.2. Breadth-first search

A common task is traversing a graph and doing some work at each vertex, e.g.

- A web crawler for a search engine (a vertex is a page, and an edge is a hyperlink). Follow all the links you can, and retrieve every page, and add it to your search index. Don't bother revisiting pages that you've already visited.
- Path finding. To find a path from one vertex v_0 to some other vertex v_1: start at v_0 and traverse the graph. Whenever you follow an edge and reach a vertex you haven't seen before, remember the path that takes you there. Stop when you reach v_1.
- Component finding. Assign each disconnected component of a graph a different colour.

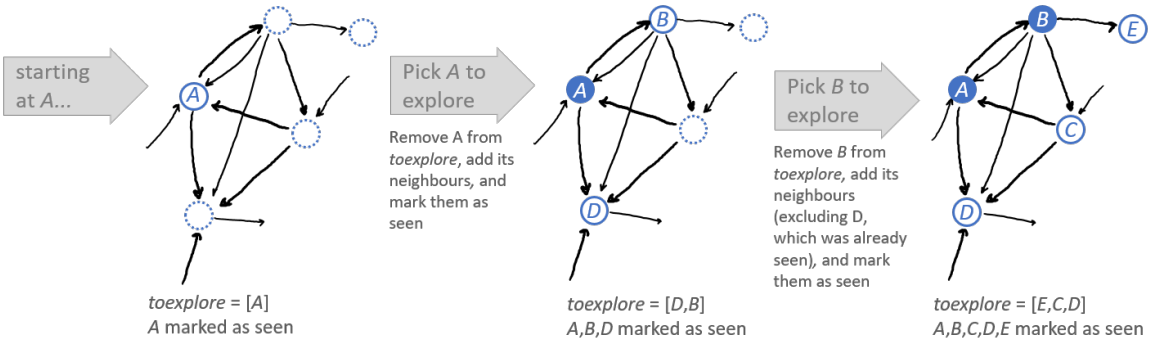
GENERAL IDEA

Whenever you visit a vertex, look at all its neighbours, and mark them as worth exploring. The *neighbours* of a vertex v are the vertices you can reach from v, i.e.

$$\text{in a directed graph,} \quad \text{neighbours}(v) = \{w \in V : v \to w\}$$
$$\text{in an undirected graph,} \quad \text{neighbours}(v) = \{w \in V : v - w\}.$$

Keep on visiting vertices-worth-exploring until you have nothing left to explore.

There is a problem: getting stuck in an infinite loop. In the example below, A is B's neighbour and B is A's neighbour, and we don't want each to keep adding the other. To prevent this, let's store a seen flag with each vertex, and set it to True to indicate that we don't need to look at that vertex again, either because we've already visited it or because it's already in the list of vertices to explore.

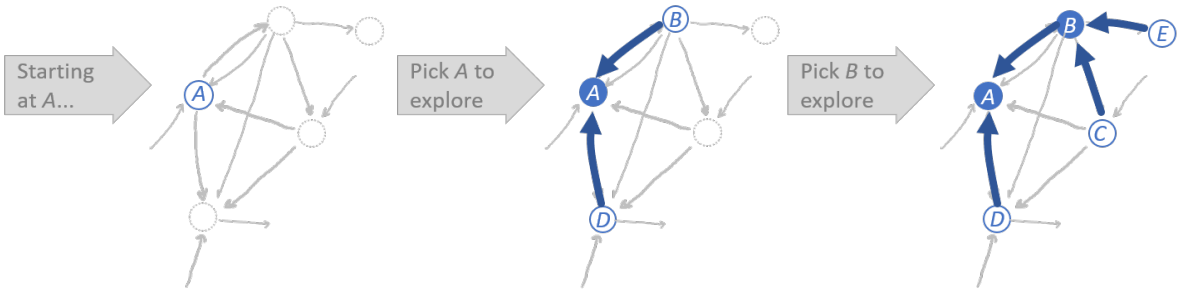


IMPLEMENTATION

This implementation uses a Queue to store the list of vertices waiting to be explored.

```
1  # Visit all the vertices in g reachable from start vertex s
2  def bfs(g, s):
3      for v in g.vertices:
4          v.seen = False
5      toexplore = Queue([s])  # a Queue initially containing a single element
6      s.seen = True
7
8      while not toexplore.is_empty():
9          v = toexplore.popright()   # Now visiting vertex v
10         for w in v.neighbours:
11             if not w.seen:
12                 toexplore.pushleft(w)
13                 w.seen = True
```

With a small tweak, we can adapt this code to find a path between a pair of nodes. All it takes is keeping track of how we discovered each vertex. Here's a picture, then the code.

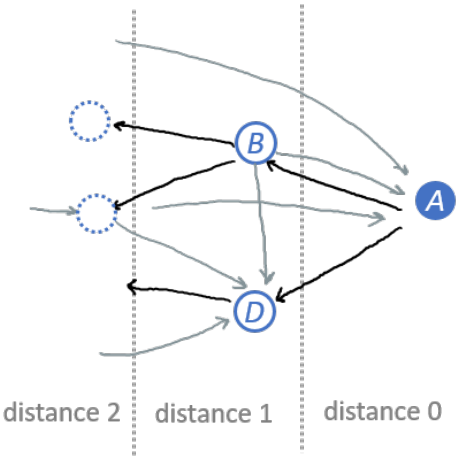



```
1  # Find a path from s to t, if one exists
2  def bfs_path(g, s, t):
3      for v in g.vertices:
4          v.seen = False
5          v.come_from = None
6      s.seen = True
7      toexplore = Queue([s])
8
9      # Traverse the graph, visiting everything reachable from s
10     while not toexplore.is_empty():
11         v = toexplore.popright()
12         for w in v.neighbours:
13             if not w.seen:
14                 toexplore.pushleft(w)
15                 w.seen = True
16                 w.come_from = v
17
18     # Reconstruct the full path from s to t, working backwards
19     if t.come_from is None:
20         return None  # there is no path from s to t
21     else:
22         path = [t]
23         while path[0].come_from != s:
24             path.prepend(path[0].come_from)
25         path.prepend(s)
26         return path
```

ANALYSIS

Running time. In bfs, (a) line 4 is run for every vertex which takes $O(V)$; (b) lines 9–10 are run at most once per vertex, since the seen flag ensures that each vertex enters toexplore at most once, which takes $O(V)$; (c) line 11 is run for every edge out of every vertex that is visited, which takes $O(E)$. Thus the total running time is $O(V + E)$.

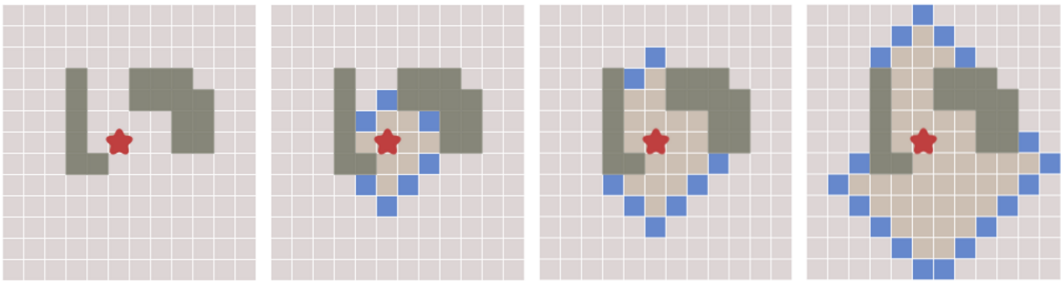
Shortest paths. The bfs_path algorithm finds the shortest path from s to t. To understand why, and what's special about the Queue, here's the same graph as before but redrawn so that vertex A is on the right, and the other vertices are arranged by their distance from A. (The *distance* from one vertex v to another vertex w is the length of the shortest path from v to w.)



If you rotate this diagram 90°anticlockwise, you can see why the algorithm is called 'breadth-first search'.

By using a Queue for toexplore (pushing new vertices on the left, popping vertices from the right), we end up exploring the graph in order of distance from the start vertex — and every come_from arrow points from a vertex at distance $d+1$ to a vertex at distance d from the start.

This gives us another way to interpret the bfs algorithm: keep track of the 'disc' of vertices that are distance $\leq d$ from the start, then grow the disc by adding the 'frontier' of vertices at distance $d+1$, and so on. What's magic is that the bfs algorithm does this implicitly, without needing an explicit variable to store d.



In this illustration[1], we're running bfs starting from the blob in the middle. The graph has one vertex for each light grey grid cell, and edges between adjacent cells, and the black cells in the left hand panel are impassable. The next three panels show some stages in the expanding frontier.

[1] These pictures are taken from the excellent Red Blob Games blog, `http://www.redblobgames.com/pathfinding/a-star/introduction.html`

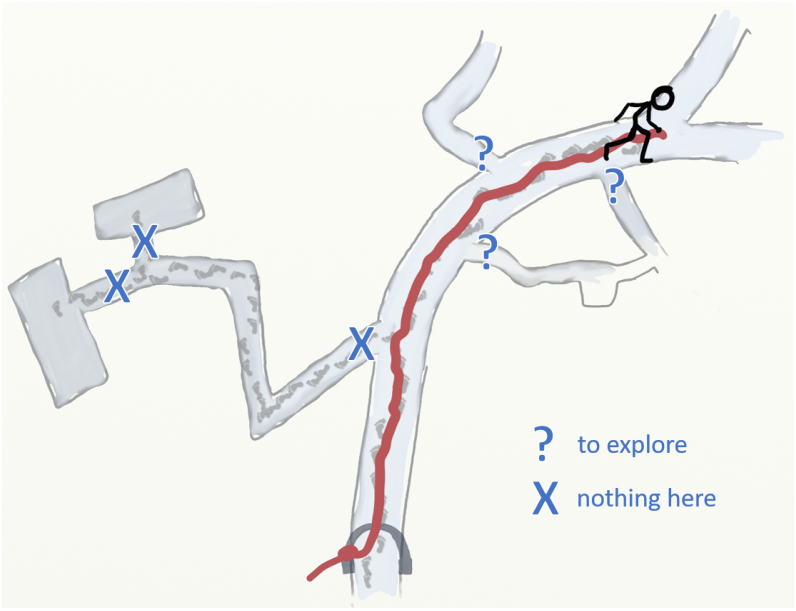
5.3. Depth-first search

GENERAL IDEA

A Greek legend describes how Theseus navigated the labyrinth containing the half-human half-bull Minotaur. His lover Ariadne gave him a ball of thread, and he tied one end at the entrance, and he unwound the thread as he walked through the labyrinth seeking the Minotaur's lair. The thread gave him a path to escape, after he slew the Minotaur.

Here are the instructions that Ariadne might have given Theseus:

- Bring chalk with you. When you have a choice of path, pick one, and chalk the others with a question mark, meaning 'waiting to be explored'.
- When you come to a dead end, backtrack along the thread. While you're backtracking, chalk the paths you took with a cross, meaning 'nothing here'. Keep backtracking until you find a path that's waiting to be explored.



IMPLEMENTATION

We can use a Stack to store all the vertices waiting to be explored. Ariadne's instruction is to backtrack until you come to a path waiting to be explored — and this must be the most recently discovered among all the paths waiting to be explored. The Stack is Last-In-First-Out, so it will automatically give us the correct next vertex.

The following code is almost identical to bfs. The only difference is that it uses a Stack rather than a Queue.

```
# Visit all vertices reachable from s
def dfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Stack([s])   # a Stack initially containing a single element
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popright()   # Now visiting vertex v
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

Here is a different implementation, using recursion. Recursion means that we're using the language's call stack, rather than our own data structure. Recursive algorithms are sometimes easier to reason about, and we'll use this implementation as part of a proof in Section 5.10. See also Example Sheet 5, which asks you to think carefully about the subtle differences between dfs and dfs_recurse.

```
1  # Visit all vertices reachable from s
2  def dfs_recurse(g, s):
3      for v in g.vertices:
4          v.visited = False
5      visit(s)
6
7  def visit(v):
8      v.visited = True
9      for w in v.neighbours:
10         if not w.visited:
11             visit(w)
```

ANALYSIS

The dfs algorithm has running time $O(V + E)$, based on exactly the same analysis as for bfs in Section 5.2.

The dfs_recurse algorithm also has running time $O(V + E)$. To see this, (a) line 4 is run once per vertex, (b) line 8 is run at most once per vertex, since the visited flag ensures that visit(v) is run at most once per vertex; (c) line 9 is run for every edge out of every vertex visited.

* * *

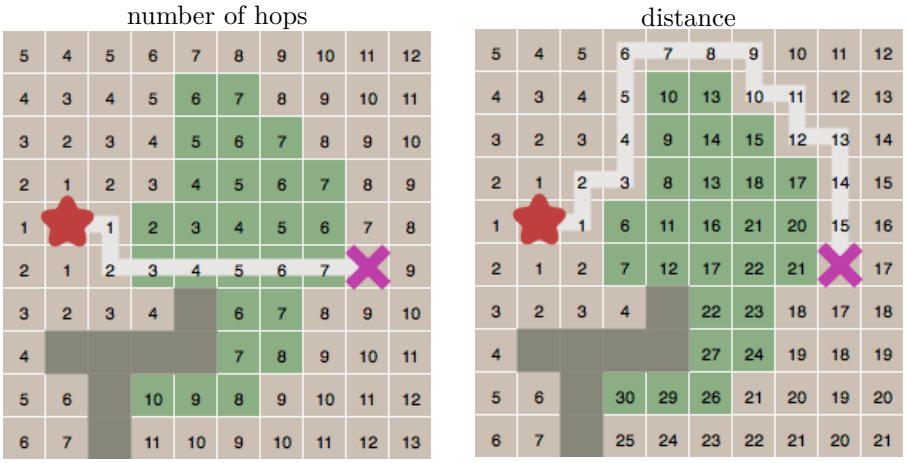
Pay close attention to the clever trick in analysing the running time of dfs_recurse. We didn't try to build up some complicated recursive formula about the running time of each call to visit, instead we used mathematical reasoning to bound the total number of times that line 8 could possibly be run during the entire execution. This is called *aggregate analysis*, and we'll see many more examples throughout the course.

5.4. Dijkstra's algorithm

In many applications it's natural to use graphs where each edge is labelled with a cost, and to look for paths with minimum cost. For example, suppose the graph's edges represent road segments, and each edge is labelled its travel time: how do we find the quickest route between two locations?

This is called the *shortest path problem*. We'll use the terms *cost* and *distance* interchangeably, and write 'distance from v_1 to v_2' to mean 'the cost of a minimum-cost path from v_1 to v_2'.

Here's an illustration[2]. These pictures show two possible paths between the blob and the cross. The left hand picture shows the number of hops from the blob; the right picture shows the distance from the blob. Here, the darkest cells can't be crossed, light cells cost 1 to cross, and darker cells cost 5.

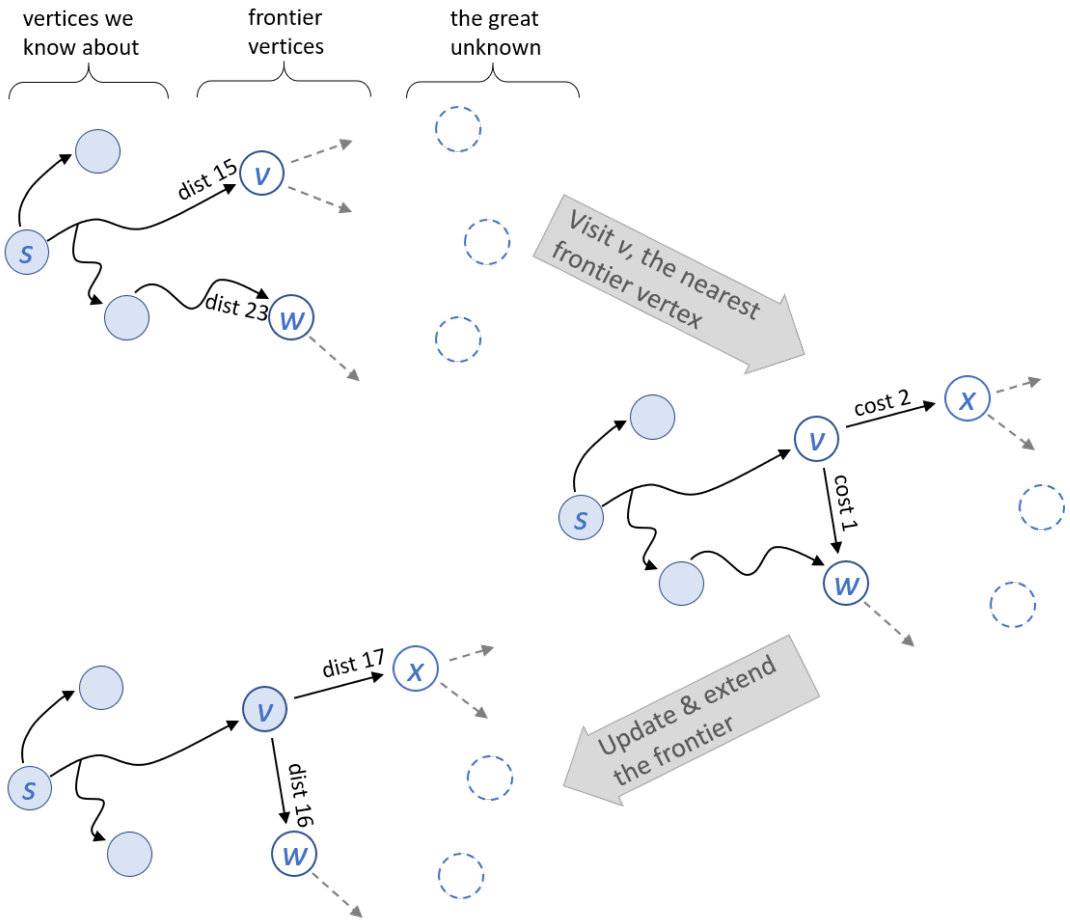


GENERAL IDEA

In breadth-first search, we visited vertices in order of how many hops they are from the start vertex. Now, let's visit vertices in order of distance from the start vertex. We'll keep track of a frontier of vertices that we're waiting to explore (i.e. the vertices whose neighbours we haven't yet examined). We'll keep the frontier vertices ordered by distance, and at each iteration we'll pick the next closest.

We might end up coming across a vertex multiple times, with different costs. If we've never come across it, just add it to the frontier. If we've come across it previously and our new path is shorter than the old path, then update its distance.

[2]Pictures taken from the Red Blob Games blog, `http://www.redblobgames.com/pathfinding/a-star/introduction.html`



PROBLEM STATEMENT

Given a directed graph where each edge is labelled with a cost ≥ 0, and a start vertex s, compute the distance from s to every other vertex.

What goes wrong in the algorithm below, and in the proof, if there are negative costs? Try to work it out yourself, before reading the answer in Section 5.5.

IMPLEMENTATION

This algorithm was invented in 1959 and is due to Dijkstra[3] (1930–2002), an influential pioneer of computer science.

Line 5 declares that toexplore is a PriorityQueue in which the key of an item v is v.distance. Line 11 iterates through all the vertices w that are neighbours of v, and retrieves the cost of the edge $v \to w$ at the same time.

See Section 4.3 for a definition of PriorityQueue. It supports inserting items, decreasing the key of an item, and extracting the item with smallest key.

```
1  def dijkstra(g, s):
2      for v in g.vertices:
3          v.distance = ∞
4      s.distance = 0
5      toexplore = PriorityQueue([s], sortkey = lambda v: v.distance)
6
7      while not toexplore.isempty():
8          v = toexplore.popmin()
9          # Assert: v.distance is the true shortest distance from s to v
10         # Assert: v is never put back into toexplore
11         for (w, edgecost) in v.neighbours:
12             dist_w = v.distance + edgecost
```

[3]Dijkstra was an idiosyncratic character famous for his way with words. Some of his sayings: "The question of whether Machines Can Think [...] is about as relevant as the question of whether Submarines Can Swim." And "If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with."

```
13              if dist_w < w.distance:
14                  w.distance = dist_w
15                  if w in toexplore:
16                      toexplore.decreasekey(w)
17                  else:
18                      toexplore.push(w)
```

Although we've called the variable v.distance, we really mean "shortest distance from s to v that we've found so far". It starts at ∞ and it decreases as we find new and shorter paths to v. Given the assertion on line 10, we could have coded this algorithm slightly differently: we could put all nodes into the priority queue in line 5, and delete lines 15, 17, and 18. It takes some work to prove the assertion...

ANALYSIS

Running time. The running time depends on how the PriorityQueue is implemented. Later in the course, we'll describe an implementation called the Fibonacci heap which for n items has $O(1)$ running time for both push() and decreasekey() and $O(\log n)$ running time for popmin(). Line 8 is run at most once per vertex (by the assertion on line 10), and lines 12–18 are run at most once per edge. So Dijkstra has running time $O(E + V \log V)$, when implemented using a Fibonacci heap.

Theorem (Correctness). *The dijkstra algorithm terminates. When it does, for every vertex v, the value v.distance it has computed is equal to the true distance from s to v. Furthermore, the two assertions are true.*

Pay close attention to whether you're dealing with abstract mathematical statements (which can be stated and proved even without an algorithm), or if you're reasoning about program execution.

Proof (of Assertion 9). Suppose this assertion fails at some point in execution, and let v be the vertex for which it first fails. Consider a shortest path from s to v. (This means the Platonic mathematical object, *not* a computed variable.) Write this path as

$$s = u_1 \to \cdots \to u_k = v$$

Let u_i be the first vertex in this sequence which has not been popped from toexplore so far at this point in execution (or, if they have all been popped, let $u_i = v$). Then,

$$\begin{aligned}
&\text{distance}(s \text{ to } v) \\
&\quad < v.\text{distance} && \text{since the assertion failed} \\
&\quad \le u_i.\text{distance} && \text{since toexplore is a PriorityQueue which had both } u_i \text{ and } v \\
&\quad \le u_{i-1}.\text{distance} + \text{cost}(u_{i-1} \to u_i) && \text{by lines 13–18 when } u_{i-1} \text{ was popped} \\
&\quad = \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i) && \text{assertion didn't fail at } u_{i-1} \\
&\quad \le \text{distance}(s \text{ to } v) && \text{since } s \to \cdots u_{i-1} \to u_i \text{ is on a shortest path } s \text{ to } v.
\end{aligned}$$

This is a contradiction, therefore the premise (that Assertion 9 failed at some point in execution) is false.

Proof (of Assertion 10). Once a vertex v has been popped, Assertion 9 guarantees that v.distance = distance(s to v). The only way that v could be pushed back into toexplore is if we found a shorter path to v (on line 13) which is impossible.

Rest of proof. Since vertices can never be re-pushed into toexplore, the algorithm must terminate. At termination, all the vertices that are reachable from s must have been visited, and popped, and when they were popped they passed Assertion 9. They can't have had v.distance changed subsequently (since it can only ever decrease, and it's impossible for it to be *less* than the true minimum distance, since the algorithm only ever looks at legitimate paths from s.) □

* * *

The proof technique was *proof by induction*. We start with an ordering (the order in which vertices were popped during execution), we assume the result is true for all earlier vertices,

and we prove it true for the next. For some graph algorithms it's helpful to order differently, e.g. by distance rather than time. Whenever you use this proof style, make sure you say explicitly what your ordering is.

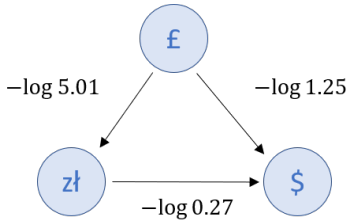
5.5. Bellman-Ford

In some applications, we have graphs where some edge weights are negative. This is useful when vertices represent states that an agent can be in, and edges represent actions that take it from one state to another; some actions might have costs and others might have rewards.

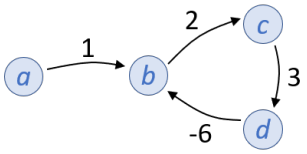
We'll write *weight* rather than cost of an edge, and *minimum weight* rather than distance between two vertices, and *minimal weight path* rather than shortest path, since the word 'distance' suggests a positive number.

Example (Currency trading). Let vertices represent currencies. Suppose we can exchange £1 for \$1.25, £1 for 5.01 zł, and 1 zł for \$0.27. If 1 unit of v_1 can be exchanged for x units of v_2, we'll put an edge from v_1 to v_2 with weight $-\log x$.

The weight of the £→zł→\$ path is $-\log 5.01 - \log 0.27 = -\log(5.01 * 0.27) = -\log 1.35$, and the weight of the direct £→\$ edge is $-\log 1.25$. Because of the log, the path weight corresponds to the net exchange rate, and because of the minus sign the minimal weight path corresponds to the most favourable exchange rate.

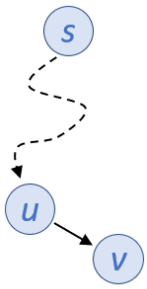


Example (Negative cycles). What's the minimum weight from a to b? By going around $b \to c \to d \to b$ again and again, the weight of the path goes down and down. This is referred to as a *negative weight cycle*, and we'd say that the minimum weight from a to b is $-\infty$.



$a \to b$: weight 1
$a \to b \to c \to d \to b$: weight 0
$a \to b \to c \to d \to b \to c \to d \to b$: weight -1

GENERAL IDEA



If we've found a path from s to u, and there is an edge $u \to v$, then we have a path from s to v. If we store the minimum weight path we've found so far in the variable minweight, then the obvious update is

$$\text{if } v.\text{minweight} > u.\text{minweight} + \text{weight}(u \to v):$$
$$v.\text{minweight} = u.\text{minweight} + \text{weight}(u \to v)$$

This update rule is known as *relaxing the edge* $u \to v$. Relaxation was at the heart of Dijkstra's algorithm (which furthermore only applied relaxation on $u \to v$ once u.minweight was the true distance).

The idea of the Bellman-Ford algorithm is simply to keep on applying this rule to all edges in the graph, over and over again, updating "best weight from s to v found so far" if a path via u gives a lower weight. The magic is that we only need to apply it a fixed number of times.

PROBLEM STATEMENT

Given a directed graph where each edge is labelled with a weight, and a start vertex s, (i) if the graph contains no negative-weight cycles reachable from s then for every vertex v compute the minimum weight from s to v; (ii) otherwise report that there is a negative weight cycle reachable from s.

IMPLEMENTATION

In this code, lines 8 and 12 iterate over all edges in the graph, and c is the weight of the edge $u \to v$. The assertion in line 10 refers to the true minimum weight among all paths from s to v, which the algorithm doesn't know yet; the assertion is just there to help us reason about how the algorithm works, not something we can actually test during execution.

```
1  def bf(g, s):
2      for v in g.vertices:
3          v.minweight = ∞ # best estimate so far of minweight from s to v
4      s.minweight = 0
5
6      repeat len(g.vertices)−1 times:
7          # relax all the edges
8          for (u,v,c) in g.edges:
9              v.minweight = min(u.minweight + c, v.minweight)
10             # Assert v.minweight >= true minimum weight from s to v
11
12     for (u,v,c) in g.edges:
13         if u.minweight + c < v.minweight:
14             throw "Negative−weight cycle detected"
```

Lines 12–14 say, in effect, "If the answer we get after $V-1$ rounds of relaxation is different to the answer after V rounds, then there is a negative-weight cycle; and vice versa."

ANALYSIS

The algorithm iterates over all the edges, and it repeats this V times, so the overall running time is $O(VE)$.

Theorem. *The algorithm correctly solves the problem statement. In case (i) it terminates successfully, and in case (ii) it throws an exception in line 14. Furthermore the assertion on line 10 is true.*

Proof (of assertion on line 10). Write $w(v)$ for the true minimum weight among all paths from s to v, with the convention that $w(v) = -\infty$ if there is a path that includes a negative-weight cycle. The algorithm only ever updates v.minweight when it has a valid path to v, therefore the assertion is true.

Proof for case (i). Pick any vertex v, and consider a minimal-weight path from s to v. Let the path be

$$s = u_0 \to u_1 \to \cdots \to u_k = v.$$

Consider what happens in successive iterations of the main loop, lines 8–10.

- Initially, u_0.minweight is correct, i.e. equal to $w(s)$ which is 0.
- After one iteration, u_1.minweight is correct. Why? If there were a lower-weight path to u_1, then the path we've got here couldn't be a minimal-weight path to v.
- After two iterations, u_2.minweight is correct.
- and so on...

We can assume (without loss of generality) that this path has no cycles—if it did, the cycle would have weight ≥ 0 by assumption, so we could cut it out. So it has at most $|V|-1$ edges, so after $|V|-1$ iterations v.minweight is correct.

Thus, by the time we reach line 12, all vertices have the correct minweight, hence the test on line 13 never goes on to line 14, i.e. the algorithm terminates without an exception.

Proof of (ii). Suppose there is a negative-weight cycle reachable from s,

$$s \to \cdots \to v_0 \to v_1 \to \cdots \to v_k \to v_0$$

where

$$\text{weight}(v_0 \to v_1) + \cdots + \text{weight}(v_k \to v_0) < 0.$$

If the algorithm terminates without throwing an exception, then all these edges pass the test in line 13, i.e.

$$\begin{aligned} v_0.\mathsf{minweight} + \text{weight}(v_0 \to v_1) &\geq v_1.\mathsf{minweight} \\ v_1.\mathsf{minweight} + \text{weight}(v_1 \to v_2) &\geq v_2.\mathsf{minweight} \\ &\vdots \\ v_k.\mathsf{minweight} + \text{weight}(v_k \to v_0) &\geq v_0.\mathsf{minweight} \end{aligned}$$

Putting all these equations together,

$$v_0.\mathsf{minweight} + \text{weight}(v_0 \to v_1) + \cdots + \text{weight}(v_k \to v_0) \geq v_0.\mathsf{minweight}$$

hence the cycle has weight ≥ 0. This contradicts the premise—so at least one of the edges must fail the test in line 13, and so the exception will be thrown. □

5.6. Johnson's algorithm

What if we want to compute shortest paths between *all* pairs of vertices?

- Each router in the internet has to know, for every packet it might receive, where that packet should be forwarded to. Path preferences in the Internet are based on link costs set by internet service providers. Routers send messages to each other advertising which destinations they can reach and at what cost. The Border Gateway Protocol (BGP) specifies how they do this. It is a distributed path-finding algorithm, and it is a much bigger challenge than computing paths on a single machine.
- The *betweenness centrality* of an edge is defined to be the number of shortest paths that use that edge, over all the shortest paths between all pairs of vertices in a graph. (If there are n shortest paths between a pair of vertices, count each of them as contributing $1/n$.) The betweenness centrality is a measure of how important that edge is, and it's used for summarizing the shape of e.g. a social network. To compute it, we need shortest paths between all pairs of vertices.

GENERAL IDEA

If all edge weights are ≥ 0, we can just run Dijkstra's algorithm V times, once from each vertex. This has running time

$$V \cdot O(E + V \log V) = O(VE + V^2 \log V).$$

If some edge weights are < 0, we could run Bellman-Ford from each vertex, which would have running time

$$V \cdot O(VE) = O(V^2 E).$$

But there is a clever trick, discovered by Donald Johnson in 1977, whereby we can run Bellman-Ford once, then run Dijkstra once from each vertex, then run some cleanup for every pair of vertices. The running time is therefore

$$O(VE) + O(VE + V^2 \log V) + O(V^2) = O(VE + V^2 \log V)$$

It's as if we cope with negative edge weights for free!

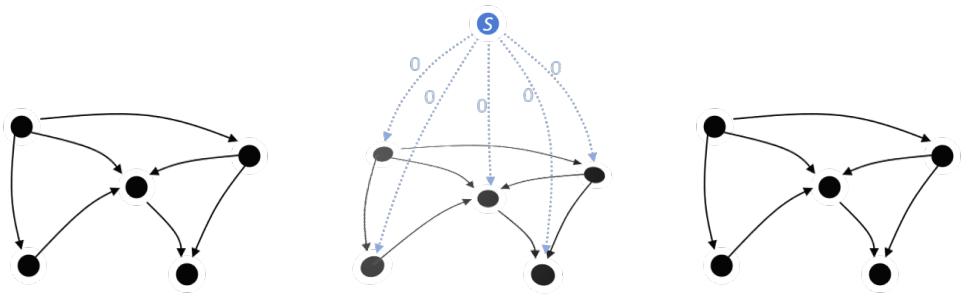
The algorithm works by constructing an extra 'helper' graph, running a computation in it, and applying the results of the computation to the original problem. This is a common pattern, and we'll see it again in Section 6.1.

PROBLEM STATEMENT

Given a directed graph where each edge is labelled with a weight, (i) if the graph contains no negative-weight cycles then for every pair of vertices compute the weight of the minimal-weight path between those vertices; (ii) if the graph contains a negative-weight cycle then detect that this is so.

IMPLEMENTATION AND ANALYSIS

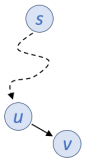
1. The helper graph. First build a helper graph, as shown below. Run Bellman-Ford on the helper graph, and let the minimum weight from s to v be d_v. (The direct path $s \to v$ has weight 0, so obviously $d_v \leq 0$. But if there are negative-weight edges in the graph, some vertices will have $d_v < 0$.) If Bellman-Ford reports a negative-weight cycle, then stop.



original graph, with edge weights $w(u \to v)$

helper graph, with an extra vertex s and zero-weight edges $s \to v$ for all vertices v

a tweaked graph with modified edge weights $w'(u \to v)$



2. The tweaked graph. Define a tweaked graph which is like the original graph, but with different edge weights:

$$w'(u \to v) = d_u + w(u \to v) - d_v.$$

CLAIM: in this tweaked graph, every edge has $w'(u \to v) \geq 0$. PROOF: The relaxation equation, applied to the helper graph, says that $d_v \leq d_u + w(u \to v)$, therefore $w'(u \to v) \geq 0$.

3. Dijkstra on the tweaked graph. Run Dijkstra's algorithm V times on the tweaked graph, once from each vertex. (We've ensured that the tweaked graph has edge weights ≥ 0, so Dijkstra terminates correctly.) CLAIM: Minimum-weight paths in the tweaked graph are the same as in the original graph. PROOF: Pick any two vertices p and q, and any path between them

$$p = v_0 \to v_1 \to \cdots \to v_k = q.$$

What weight does this path have, in the tweaked graph and in the original graph?

This algebraic trick is called a *telescoping sum*.

$$\begin{aligned}
&\text{weight in tweaked graph} \\
&\quad = d_p + w(v_0 \to v_1) - d_{v_1} + d_{v_1} + w(v_1 \to v_2) - d_{v_2} + \cdots \\
&\quad = d_p + w(v_0 \to v_1) + w(v_1 \to v_2) + \cdots + w(v_{k-1} \to v_k) - d_q \\
&\quad = \text{weight in original graph} + d_p - d_q.
\end{aligned}$$

Since $d_p - d_q$ is the same for every path from p to q, the ranking of paths is the same in the tweaked graph as in the original graph (though of course the weights are different).

4. Wrap up. We've just shown that

$$\begin{array}{l} \text{min weight} \\ \text{from } p \text{ to } q \\ \text{in original graph} \end{array} = \begin{array}{l} \text{min weight} \\ \text{from } p \text{ to } q \\ \text{in tweaked graph} \end{array} - d_p + d_q$$

which solves the problem statement.

5.7. All-pairs shortest paths with matrices

There is another algorithm to find shortest paths between all pairs of vertices, which is based entirely on algebra with barely any thought about graphs. Its running time is $O(V^3 \log V)$. This is worse than Johnson's algorithm, but it's very simple to implement. And it's a nice example of what you can do with clever notation, which is a good trick to have up your sleeve.

GENERAL IDEA

The art of dynamic programming is figuring out how to express our problem in a way that has easier subproblems. Sometimes, we can achieve this by turning our original problem into something that seems harder. In this case,

> Let $M^{(\ell)}$ be a $V \times V$ matrix, where $M^{(\ell)}_{ij}$ is the minimum weight among all paths from i to j that have ℓ or fewer edges.

We can write out a simple equation for $M^{(\ell)}$ in terms of $M^{(\ell-1)}$, and this leads directly to an algorithm for computing $M^{(\ell)}$. If we pick ℓ big enough (at least the maximum number of edges in any shortest path) then we've solved the original problem.

PROBLEM STATEMENT

(Same as for Johnson's algorithm.) Given a directed graph where each edge is labelled with a weight, (i) if the graph contains no negative-weight cycles then for every pair of vertices compute the weight of the minimal-weight path between those vertices; (ii) if the graph contains a negative-weight cycle then detect that this is so.

IMPLEMENTATION

Let $n = |V|$ be the number of edges, and define the $n \times n$ matrix W by

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight}(i \to j) & \text{if there is an edge } i \to j \\ \infty & \text{otherwise.} \end{cases}$$

Then, thinking carefully through the definition of $M^{(n)}$, $M^{(1)} = W$ and

The notation $x \wedge y$ means $\min(x, y)$.

$$\begin{aligned} M^{(\ell)}_{ij} &= M^{(\ell-1)}_{ij} \wedge \left[\left(M^{(\ell-1)}_{i1} + W_{1j}\right) \wedge \left(M^{(\ell-1)}_{i2} + W_{2j}\right) \wedge \cdots \wedge \left(M^{(\ell-1)}_{in} + W_{nj}\right) \right] \\ &= \left(M^{(\ell-1)}_{i1} + W_{1j}\right) \wedge \left(M^{(\ell-1)}_{i2} + W_{2j}\right) \wedge \cdots \wedge \left(M^{(\ell-1)}_{in} + W_{nj}\right). \end{aligned}$$

The first line expresses "To go from i to j in $\le \ell$ hops, you could either go in $\le \ell - 1$ hops, or you could go from i to some other node k in $\le \ell - 1$ hops, then take the edge $k \to j$." The second line is simple algebra, $M^{(\ell-1)}_{ij} = M^{(\ell-1)}_{ij} + W_{jj}$ because $W_{jj} = 0$.

This is just like regular matrix multiplication

$$[A\,B]_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj}$$

except it uses $+$ instead of multiplication and $\wedge$ instead of addition. Let's write it $M^{(\ell)} = M^{(\ell-1)} \otimes W$. The full algorithm is like Bellman-Ford:

```
Let M^(1) = W
Compute M^(V-1) and M^(V), using M^(ℓ) = M^(ℓ-1) ⊗ W
If M^(V-1) == M^(V):
    return M^(V-1) # this matrix consists of minimum weights
else:
    throw "negative weight cycle detected"
```

ANALYSIS

Correctness. We've explained why $M_{ij}^{(\ell)}$ is the minimum weight among all paths of length $\leq \ell$. The proof that lines 3–6 are correct is almost identical to the proof for Bellman-Ford.

Running time. As with regular matrix multiplication, it takes V^3 operations to compute $\otimes$, so the total running time is $O(V^4)$. There is a cunning trick to reduce the running time. Let's illustrate with $V = 10$. Rather than applying $\otimes$ 8 times to compute $M^{(9)}$, we can repeatedly square:

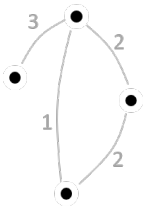
$$\begin{aligned} M^{(1)} &= W \\ M^{(2)} &= M^{(1)} \otimes M^{(1)} \\ M^{(4)} &= M^{(2)} \otimes M^{(2)} \\ M^{(8)} &= M^{(4)} \otimes M^{(4)} \\ M^{(16)} &= M^{(8)} \otimes M^{(8)} \\ &= M^{(9)} \quad \text{if there are no negative-weight cycles.} \end{aligned}$$

This trick gives overall running time $O(V^3 \log V)$.

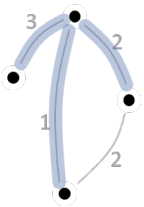
5.8. Prim's algorithm

Given a connected undirected graph with edge weights, a *minimum spanning tree* (MST) is a tree that 'spans' the graph i.e. connects all the vertices, and which has minimum weight among all spanning trees. (The *weight* of a tree is just the sum of the weights of its edges.)

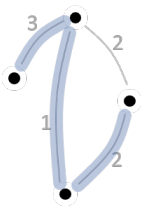
See Section 5.1 for the definition of 'connect' and 'tree'.



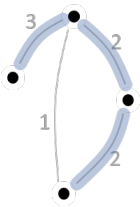
undirected graph with edge weights



spanning tree of weight 6



spanning tree of weight 6

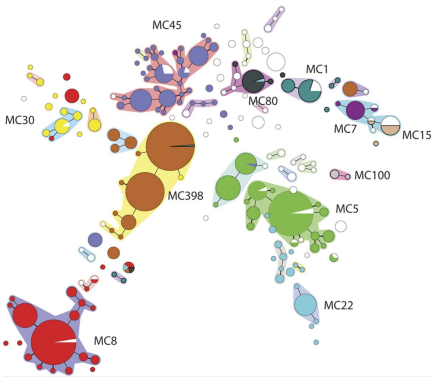


spanning tree of weight 7

APPLICATIONS

- The MST problem was first posed and solved by the Czech mathematician Borůvka in 1926, motivated by a network planning problem. His friend, an employee of the West Moravian Powerplants company, put to him the question: if you have to build an electrical power grid to connect a given set of locations, and you know the costs of running cabling between locations, what is the cheapest power grid to build?

- Minimal spanning trees are a useful tool for exploratory data analysis. In this illustration from bioinformatics[4], each vertex is a genotype of *Staphylococcus aureus*, and the size shows the prevalance of that genotype in the study sample. Let the there be edges between all genotypes, weighted according to edit distance. The illustration shows the MST, after some additional high-weight edges are removed.

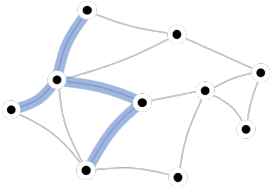


The 'edit distance' between two strings is a measure of how different they are. See Section 1.2.1.

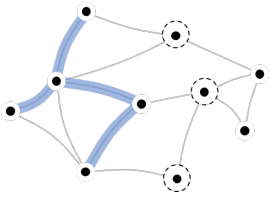
GENERAL IDEA

We'll build up the MST greedily. Suppose we've already built a tree containing some of the vertices (start it with just a single vertex, chosen arbitrarily). Look at all the edges between the tree we've built so far and the other vertices that aren't part of the tree, pick the edge of lowest weight among these and add it to the tree, then repeat.

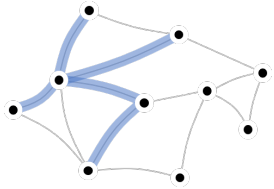
This greedy algorithm will certainly give us a spanning tree. To prove that it's a MST takes some more thought.



a tree build up with four edges so far



three candidate vertices to add next



pick the cheapest of the four connecting edges and add it to the tree

[4] From *Multiple-Locus Variable Number Tandem Repeat Analysis of Staphylococcus Aureus*, Schouls et al., PLoS ONE 2009.

PROBLEM STATEMENT

Given a connected undirected graph with edge weights, construct an MST.

IMPLEMENTATION

We don't need to recompute the nearby vertices every iteration. Instead we can use a structure very similar to Dijkstra's algorithm for shortest paths: store a 'frontier' of vertices that are neighbours of the tree, and update it each iteration. This algorithm is due to Jarnik (1930), and independently to Prim (1957) and Dijkstra (1959). When the algorithm terminates, an MST is formed from the edges

$$\{v - v.\text{come\_from} \ : \ v \in V, v \neq s\}.$$

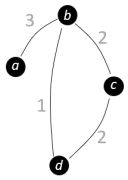
Compared to Dijkstra's algorithm, we need some extra lines to keep track of the tree (labelled +), and two modified lines (labelled ×) because here we're interested in 'distance from the tree' whereas Dijkstra is interested in 'distance from the start node'. The start vertex s can be chosen arbitrarily.

```
1   def prim(g, s):
2       for v in g.vertices:
3           v.distance = ∞
4 +         v.in_tree = False
5 +     s.come_from = None
6       s.distance = 0
7       toexplore = PriorityQueue([s], lambda v : v.distance)
8
9       while not toexplore.isempty():
10          v = toexplore.popmin()
11 +        v.in_tree = True
12          # Let t be the graph made of vertices with in_tree=True,
13          # and edges {w—w.come_from, for w in g.vertices excluding s}.
14          # Assert: t is part of an MST for g
15          for (w, edgeweight) in v.neighbours:
16 ×            if (not w.in_tree) and edgeweight < w.distance:
17 ×                w.distance = edgeweight
18 +                w.come_from = v
19                  if w in toexplore:
20                      toexplore.decreasekey(w)
21                  else:
22                      toexplore.push(w)
```

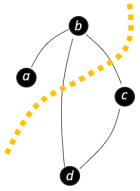
ANALYSIS

Running time. It's easy to check that Prim's algorithm terminates. It is nearly identical to Dijkstra's algorithm, and exactly the same analysis of running time applies: is $O(E+V \log V)$, assuming the priority queue is implemented using a Fibonacci heap.

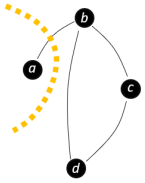
Correctness. To prove that Prim's algorithm does indeed find an MST (and for many other problems to do with constructing networks on top of graphs) it's helpful to make a definition. A *cut* of a graph is an assignment of its vertices into two non-empty sets, and an edge is said to *cross* the cut if its two ends are in different sets.



an undirected graph with edge weights



a cut into $\{a, b\}$ and $\{c, d\}$, with two edges crossing the cut

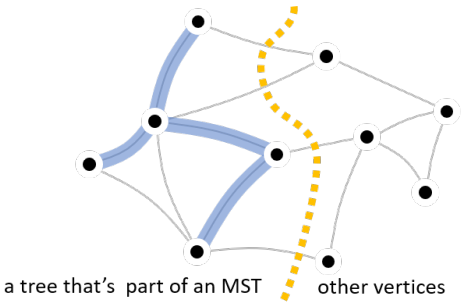


a cut into $\{a\}$ and $\{b, c, d\}$, with one edge crossing the cut

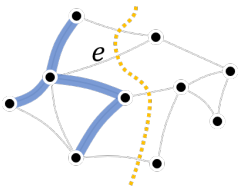
Prim's algorithm builds up a tree, adding edges greedily. By the following theorem, Prim's algorithm produces an MST.

Theorem. *If we have a tree which is part of an MST, and we add to it the min-weight edge across the cut separating the tree from the other vertices, then the result is still part of an MST.*

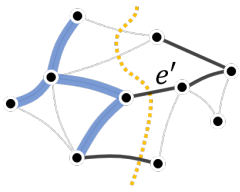
This theorem is pure maths, not a statement about program execution.



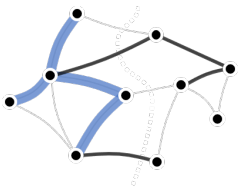
Proof. Let f be the tree, and let $\bar{f}$ be an MST that f is part of (the condition of the theorem requires that such an $\bar{f}$ exists). Let e be the a minimum weight edge across the cut. We want to show that there is an MST that includes $f \cup \{e\}$. If $\bar{f}$ includes edge e, we are done.



the tree f



a MST $\bar{f}$



a different MST $\hat{f}$

Suppose then that $\bar{f}$ doesn't contain e. Let u and v be the vertices at either end of e, and consider the path in $\bar{f}$ between u and v. (There must be such a path, since $\bar{f}$ is a spanning tree, i.e. it connects all the vertices.) This path must cross the cut (since its ends are on different sides of the cut). Let e' be an edge in the path that crosses the cut. Now, let $\hat{f}$ be like $\bar{f}$ but with e added and e' removed.

It's easy to see that weight$(\hat{f}) \leq$ weight$(\bar{f})$: e is a min-weight edge in the cut, so weight$(e) \leq$ weight(e'). CLAIM: $\hat{f}$ is also a spanning tree. If this claim is true, then $\hat{f}$ is an MST including $f \cup \{e\}$, and the theorem is proved.

PROOF OF CLAIM. This is left to an example sheet. □

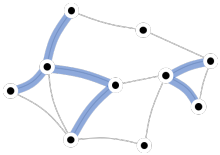
5.9. Kruskal's algorithm

Another algorithm for finding a minimum spanning tree is due to Kruskal (1956). It makes the same assumptions as Prim's algorithm. Its running time is worse. It does however produce intermediate states which can be useful.

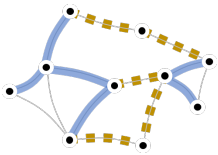
GENERAL IDEA

Kruskal's algorithm maintains a 'forest'. Look back at Section 5.1 for the definition.

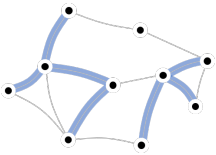
Kruskal's algorithm builds up the MST by agglomerating smaller subtrees together. At each stage, we've built up some fragments of the MST. The algorithm greedily chooses two fragments to join together, by picking the lowest-weight edge that will join two fragments.



four tree fragments have been found so far, including two trees that each consist of a single vertex



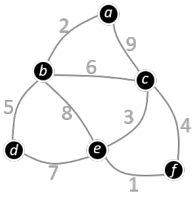
five candidate edges that would join two fragments



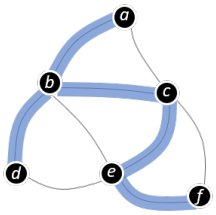
pick the cheapest of the five candidate edges, and add it, thereby joining two fragments

APPLICATION

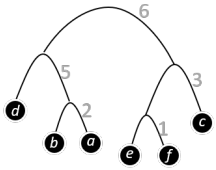
If we draw the tree fragments another way, the operation of Kruskal's algorithm looks like clustering, and its intermediate stages correspond to a classication tree:



an undirected graph with edge weights

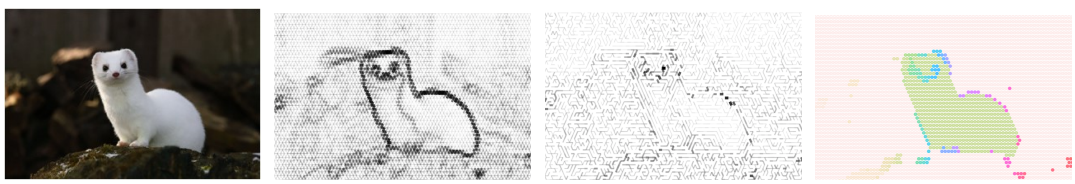


the MST found by Kruskal's algorithm



draw each fragment as a subtree, and draw arcs when two fragments are joined

This can be used for image segmentation. Here we've started with an image, put vertices on a hexagonal grid, added edges between adjacent vertices, given low weight to edges where the vertices have similar colour and brightness, run Kruskal's algorithm to find an MST, split the tree into clusters by removing a few of the final edges, and coloured vertices by which cluster they belong to.



PROBLEM STATEMENT

(Same as for Prim's algorithm.) Given a connected undirected graph with edge weights, construct an MST.

IMPLEMENTATION

This code uses a data structure called a *disjoint set*. This is used to keep track of a collection of disjoint sets (sets with no common elements), also known as a partition. We'll learn more about it in Section 7.4. Here, we're using it to keep track of which vertices are in which fragment. Initially (lines 4–5) every vertex is in its own fragment. As the algorithm proceeds, it considers each edge in turn, and looks up the vertex-sets containing the start and the end of the edge. If they correspond to different fragments, it's safe to join the fragments, i.e. merge the two sets (line 13).

Lines 6 and 8 are used to iterate through all the edges in the graph in order of edge weight, lowest edge weight first.

```
1  def kruskal(g):
2      tree_edges = []
3      partition = DisjointSet()
4      for v in g.vertices:
5          partition.addsingleton(v)
6      edges = sorted(g.edges, sortkey = lambda u,v,edgeweight: edgeweight)
7
8      for (u,v,edgeweight) in edges:
9          p = partition.getsetwith(u)
10         q = partition.getsetwith(v)
11         if p != q:
12             tree_edges.append((u,v))
13             partition.merge(p, q)
14             # Let f be the forest made up of edges in tree_edges.
15             # Assert: f is part of an MST
16             # Assert: f has one connected component per set in partition
17
18     return tree_edges
```

ANALYSIS

Running time. The running time of Kruskal's algorithm depends on how DisjointSet is implemented. We'll see in Section 7.4 that all the operations on DisjointSet can be done in $O(1)$ time[5]. The total cost is $O(E \log E)$ for the sort on line 6; $O(E)$ for iterating over edges in lines 8–11; and $O(V)$ for lines 12–13, since there can be at most V merges. So the total running time is $O(E \log E)$.

The maximum possible number of edges in an undirected graph is $V(V-1)/2$, and the minimum number of edges in a connected graph is $V-1$, so $\log E = \Theta(\log V)$, and so the running time can be written $O(E \log V)$.

Correctness. To prove that Kruskal's algorithm finds an MST, we apply the theorem used for the proof of Prim's algorithm, as follows. When the algorithm merges fragments p and q, consider the cut of all vertices into p versus not-p; the algorithm picks a minimum-weight edge across this cut, and so by the theorem we've still got an MST.

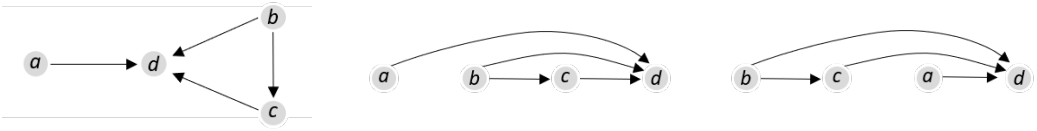
[5]This is a white lie. The actual complexity is $O(\alpha_n)$ for a DisjointSet with n elements, where α_n is a function that grows extraordinarily slowly.

5.10. Topological sort

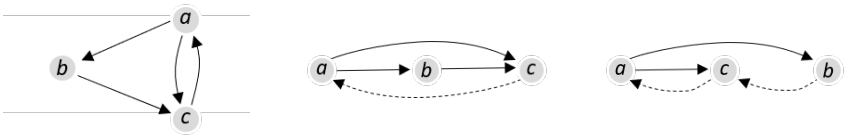
This problem is in the same general category as finding a minimal spanning tree: they are all problems of discovering organisation within a graph.

A directed graph can be used to represent ordering or preferences. We might then like to find a *total ordering* (also known as a *linear ordering* or *complete ranking*) that's compatible.

Here's a simple graph and two possible total orderings.



Does there exist a total order? If the graph has cycles, then no.

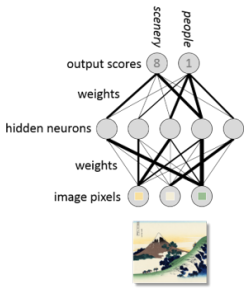


Don't get muddled by the word 'acyclic'. A DAG doesn't have to be a tree, and it might have multiple paths between vertices. The top row of graphs on this page are all DAGs.

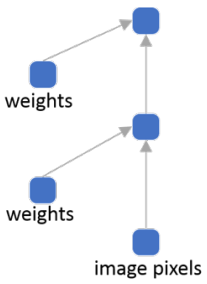
Recall the definition of a *directed acyclic graph* (DAG). A *cycle* is a path from a vertex back to itself, following the edge directions, and a directed graph is called *acyclic* if it has no cycles. We will see that, in a DAG, a total ordering can always be found.

APPLICATIONS

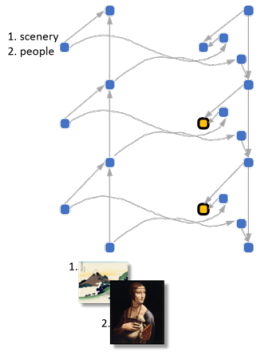
- Deep learning systems like TensorFlow involve writing out the learning task as a collection of computational steps, each of which depends on the answers of some of the preceding steps. Write $v_1 \to v_2$ to mean "Step v_2 depends on the output of v_1." If the computation graph is a DAG, then we can find an order in which to run all the computational steps. If it's not a DAG, then there is a circular dependency.



an image classifier, implemented with a deep neural network; the magic is finding good link weights

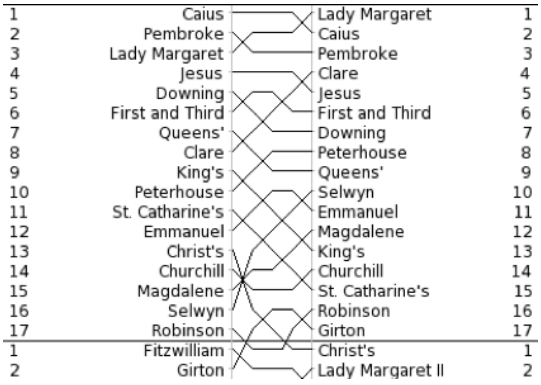


a DAG computational graph depicting how the classifier operates



a DAG computational graph for computing how the weights should be updated, based on a training dataset of pre-labelled images

- The river Cam isn't wide enough for a conventional race between all the rowing boats that want to compete. Instead, the Bumps evolved, as a way of ranking boats based on pairwise comparisons. The competition takes place over four days. On the first day, the boats start out spaced evenly along a stretch of the river, in order of last year's ranking. They start rowing all at the same time, and each boat tries to catch up—bump—the boat ahead. If this happens, then both boats move to the side of the river and withdraw from this day's race, and they switch their starting positions for the next day's race. Four days of the Bumps give us a set of pairwise comparisons: if boat v_1 bumps v_2, then we know v_1 is faster than v_2. Here are the men's bumps from May 2016. What are the total orderings consistent with this data, if any?



If the pairwise comparisons don't form a DAG, then it's impossible to find a total order—but we can still look for a order that's mostly consistent. There are many applications in machine learning with this flavour, where we think there is some hidden order or structure which we have to reconstruct based on noisy data.

GENERAL IDEA

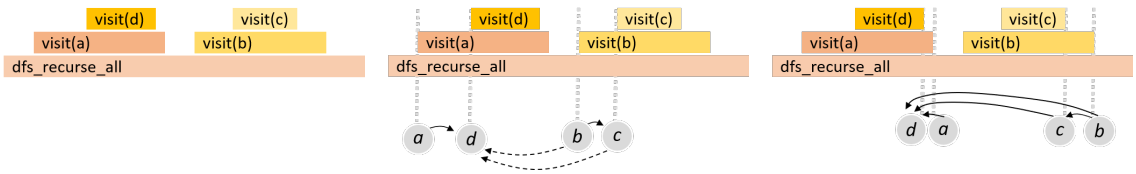
Recall depth-first search. After reaching a vertex v, it visits all v's children and other descendants. We want v to appear earlier in the ordering than all its descendants. So, can we use depth-first search to find a total ordering?

Here again is the depth-first search algorithm. This is dfs_recurse from Section 5.3, but modified so that it visits the entire graph (rather than just the part reachable from some given start vertex).

```
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)  # start dfs from v

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)
```

A standard way to visualise program execution is with a *flame chart*. Time goes on the horizontal axis, each function call is shown as a rectangle, and if function f calls function g then g is drawn above g. Here is a flame chart for the graph at the beginning of this section.



If we order vertices by when the algorithm first visits them, it turns out not to be a total order. A better guess is to order vertices by when visit(v) *returns*.

PROBLEM STATEMENT

Given a directed acyclic graph (DAG), return a total ordering of all its vertices, such that if $v_1 \to v_2$ then v_1 appears before v_2 in the total order.

ALGORITHM

This algorithm is due to Knuth. It is based on dfs_recurse_all, with some extra lines (labelled +). These extra lines build up a linked list for the rankings, as the algorithm visits and leaves each vertex.

```
1  def toposort(g):
2      for v in g.vertices:
3          v.visited = False
4          # v.colour = 'white'
5  +   totalorder = []  # an empty list
6      for v in g.vertices:
7          if not v.visited:
8              visit(v, totalorder)
9  +   return totalorder
10
11 def visit(v, totalorder):
12     v.visited = True
13     # v.colour = 'grey'
14     for w in v.neighbours:
15         if not w.visited:
16             visit(w, totalorder)
17 +   totalorder.prepend(v)
18     # v.colour = 'black'
```

This listing also has some commented lines which aren't part of the algorithm itself, but which are helpful for arguing that the algorithm is correct. They're a bit like assert statements: they're there for our understanding of the algorithm, not for its execution.

ANALYSIS

Running time. We haven't changed anything substantial from dfs_recurse so the analysis in Section 5.3 still applies: the running time is $O(V + E)$.

Theorem (Correctness). *The toposort algorithm terminates and returns totalorder which solves the problem statement.*

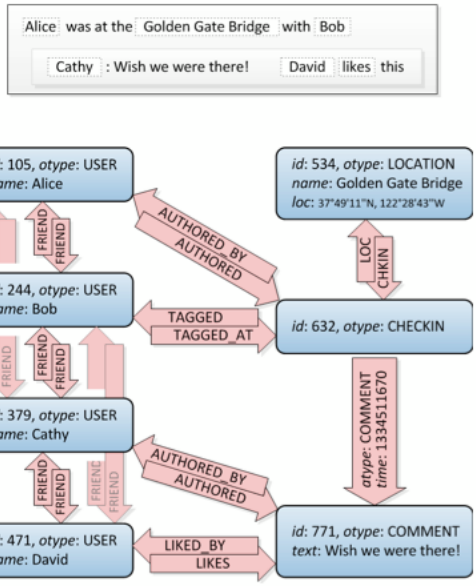
Proof. Pick any edge $v_1 \to v_2$. We want to show that v_1 appears before v_2 in totalorder. It's easy to see that every vertex is visited exactly once, and on that visit (1) it's coloured grey, (2) some stuff happens, (3) it's coloured black. Let's consider the instant when v_1 is coloured grey. At this instant, there are three possibilities for v_2:

- v_2 is black. If this is so, then v_2 has already been prepended to the list, so v_1 will be prepended after v_2, so v_1 appears before v_2.
- v_2 is white. If this is so, then v_2 hasn't yet been visited, therefore we'll call visit(v_2) at some point during the execution of lines 14–16 in visit(v_1). This call to visit(v_2) must finish before returning to the execution of visit(v_1), so v_2 gets prepended earlier and v_1 gets prepended later, so v_1 appears before v_2.
- v_2 is grey. If this is so, then there was an earlier call to visit(v_2) which we're currently inside. The call stack corresponds to a path in the graph from v_2 to v_1. But we've picked an edge $v_1 \to v_2$, so there is a cycle, which is impossible in a DAG. This is a contradiction, so it's impossible that v_2 is grey. □

5.11. Graphs and big data

FACEBOOK

Facebook sees the world as a graph of objects and associations, their *social graph*[6]:



Facebook represents this internally with classic database tables:

id	otype	attributes
105	USER	{name: Alice}
244	USER	{name: Bob}
379	USER	{name: Cathy}
471	USER	{name: David}
534	LOCATION	{name: Golden Gate Bridge, loc: (38.9,-77.04)}
632	CHECKIN	
771	COMMENT	{text: Wish we were there!}

from_id	to_id	edge_type
105	244	FRIEND
105	379	FRIEND
105	632	AUTHORED
244	105	FRIEND
244	379	FRIEND
244	632	TAGGED_AT

Why not use an adjacency list? Some possible reasons:

- Backups! Years of bitter experience have gone into today's databases, and they are very good and reliable for mundane tasks like backing up your data. The data is too big to fit in memory, and database tables are a straightforward way to store it on disk.
- Database tables can be indexed on many keys. If I have a query like "Find all edges to or from user 379 with timestamp no older than 24 hours", and if the edges table has indexes for columns from_id and to_id and timestamp, then the query can be answered quickly. In an adjacency list representation, we'd just have to trawl through all the edges.

When you visit your home page, Facebook runs many queries on its social graph to populate the page. It needs to ensure that the common queries run very quickly, and so it has put considerable effort into indexing and caching.

Twitter also has a huge graph, with vertices for tweets and users, and edges for @mentions and follows. Like Facebook, it has optimized its graph database to give rapid answers to 'broad but shallow' queries on the graph, such as "Who is following both the tweeter and the @mentioned user?"

[6] *TAO: Facebook's Distributed Data Store for the Social Graph*, Bronson et al., Usenix 2013

GOOGLE, SPARK

Google first came to fame because they had a better search engine than anyone else. The key idea, by Brin and Page when they were PhD students at Stanford, was this: a webpage is likely to be 'good' if other 'good' webpages link to it. They built a search engine which ranked results not just by how well they matched the search terms, but also by the 'goodness' of the pages. They use the word PageRank rather than goodness, and the equation they used to define it is

$$\mathrm{PR}_v = \frac{1-\delta}{|V|} + \delta \sum_{u\,:\,u\to v} \frac{\mathrm{PR}_u}{|u.\mathsf{neighbours}|}$$

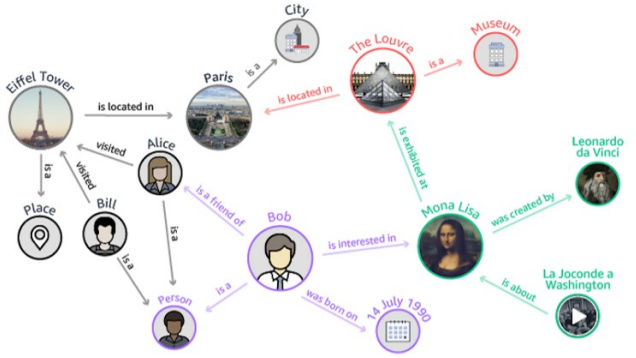
where $\delta = 0.85$ is put in as a 'dampening factor' that ensures the equations have a well-behaved unique solution.

How do we solve an equation like this, on a giant graph with one vertex for every webpage? Google said in 2013 that it indexes more than 30 trillion unique webpages, so the graph needs a cluster of machines to store it, and the computation has to be run on the cluster.

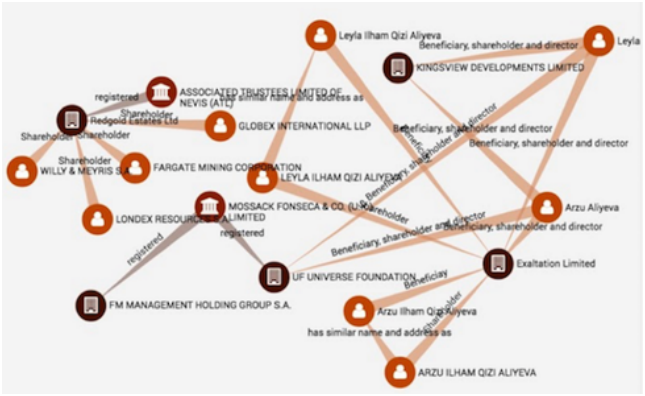
A popular platform for distributed computation (as of writing these notes in 2018) is called Spark[7]. It has a library that is tailor-made for distributed computation over graphs, and friendly tutorials.

KNOWLEDGE GRAPHS AND GRAPH DATABASES

A knowledge graph is a graph designed to capture facts about real-world objects and their relationships.



Knowledge graphs are used by Alexa and Google Assistant and so on, to hopefully be able to answer questions like "In what cities can I see art by Leonardo da Vinci?".



When the Panama Papers dataset was leaked[8], uncovering a complex network of offshore trusts, the journalists who got hold of it used a graph database called neo4j to help them understand it. You have learnt (or will learn) more about neo4j in IA/IB Databases.

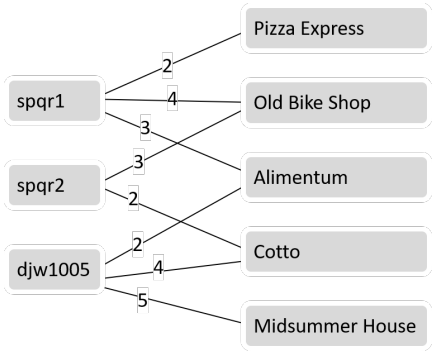
[7] http://spark.apache.org/docs/latest/graphx-programming-guide.html#pagerank
[8] https://offshoreleaks.icij.org/pages/database

6. Networks and flows

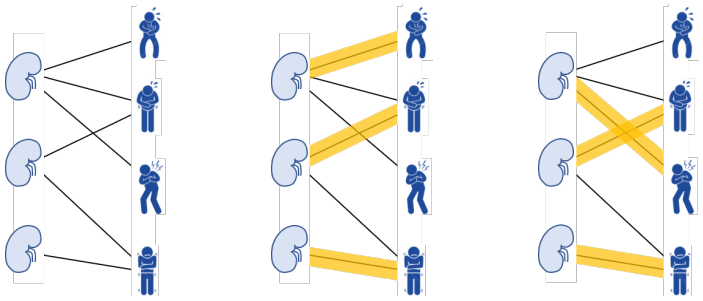
6.1. Matchings

A *bipartite graph* is one in which the vertices are split into two sets, and all the edges have one end in one set and the other end in the other set. We'll assume the graph is undirected. For example

- Vertices for medical school graduates, vertices for hospitals offering residency training, and edges for each application the medic has made to a hospital.
- Vertices for Yelp users, vertices for restaurants, and edges labelled with weights to indicate the user's rating of the restaurant.

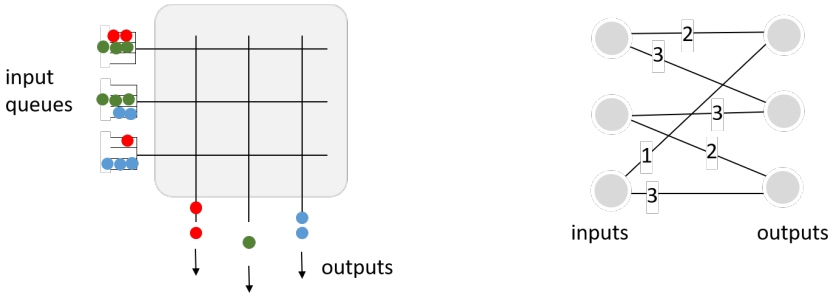


A *matching* in a bipartite graph is a selection of some or all of graph's edges, such that no vertex is connected to more than one edge in this selection. For example, kidney transplant donors and recipients, with edges to indicate compatibility. The *size* of a matching is the number of edges it contains. A *maximum matching* is one with the largest possible size. There may be several maximum matchings.



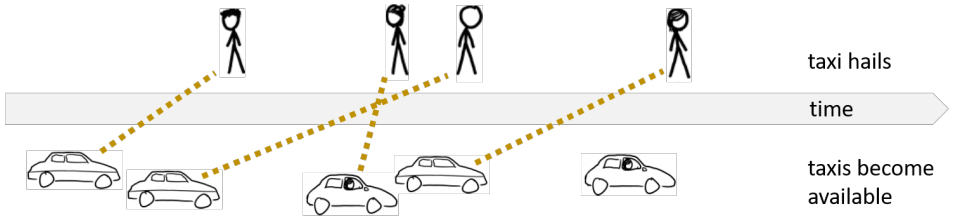
APPLICATIONS

Example (Internet switches). The silicon in the heart of an Internet router has the job of forwarding packets from inputs to outputs. Every clock tick, it can take at most one packet from each input, and it can send at most one packet to each output—in other words, it selects a matching from inputs to outputs. It turns out to be useful to weight the edges by the number of packets waiting to be sent, and to pick a matching with the highest possible total weight.



Example (Taxi scheduling). A company like Uber has to match taxis to passengers. When there are passengers who have made requests, which taxis should they get[9]? This is an example of an *online matching problem*. (As opposed to the offline matching problem, in which all the vertices and edges are known in advance.) In online problems, bad choices will have kock-on effects.

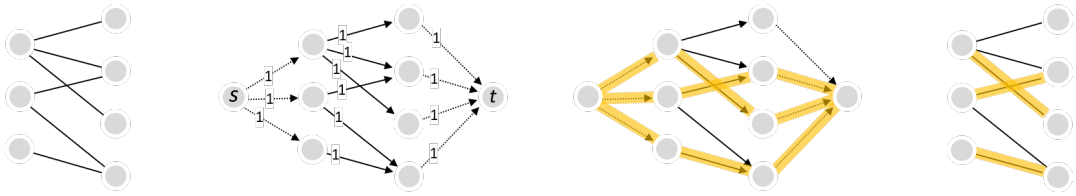
The simple greedy strategy 'pick the nearest available taxi as soon as a request is made' might lead to shortages on the perimeter of the network, or to imbalanced workloads among drivers. We could turn it into an offline problem by batching, for example 'once a minute, put edges from each waiting passenger to the ten nearest available vehicles, and look for a maximum matching'.



IMPLEMENTATION

An good way to find a maximum matching in a bipartite graph is to turn it into what looks like a harder problem, the *maximum flow problem*. **Read Sections 6.2 and 6.3 now.** The translation is as follows:

1. start with a bipartite graph
2. add a source s with edges to each left-hand vertex; add a sink with edges from each right-hand vertex; turn the original edges into directed edges from left to right; give all edges capacity 1
3. run the Ford-Fulkerson algorithm to find a maximum flow from s to t
4. interpret that flow as a matching



ANALYSIS

It's easy to not even notice that there's something that needs to be proved here. It's actually a rather subtle argument. We're relying on an equivalence between 'solution to matching problem' and 'solution to flow problem', and we have to show that the equivalence goes both ways. A question on Example Sheet 6 requires the same proof style, to relate flow problems to London tube disruptions.

Theorem. *1. The maximum matching algorithm described above terminates.*
2. It produces a matching.
3. There is no matching with larger size (i.e., it produces a maximum matching.)

Proof (of 1). The lemma in Section 6.3 on page 40 tells us that the Ford-Fulkerson algorithm terminates, since all edge capacities are integer.

Proof (of 2). Write f^* for the flow produced by Ford-Fulkerson. The lemma tells us furthermore that f^* is integer on all edges. Since the edge capacities are all 1, the flow must be 0

[9] Figure elements from Randall Munroe, `https://what-if.xkcd.com/9/` and `https://what-if.xkcd.com/93/`

or 1 on all edges. Translate f^* into a matching m^*, by simply selecting all the edges in the original bipartite graph that got $f^* = 1$. The capacity constraints on edges from s means that each left-hand vertex has either 0 or 1 flow coming in, so it must have 0 or 1 flow going out, therefore it is connected to at most one edge in m^*. Similarly, each right-hand vertex is connected to at most one edge in m^*. Therefore m^* is a matching.

Proof (of 3). Consider any other matching m. We can translate m into a flow f, in the obvious way. The translation between flows and matchings means that

$$\text{size}(m) = \text{value}(f), \qquad \text{size}(m^*) = \text{value}(f^*).$$

We know that f^* is a maximum flow, therefore

$$\text{value}(f) \leq \text{value}(f^*) \qquad \Longrightarrow \qquad \text{size}(m) \leq \text{size}(m^*)$$

Hence m^* is a maximum matching. □

6.2. Max-flow min-cut theorem

To describe a transportation network, we can use a directed graph with edge weights: vertices for the junctions, edges for the roads or railway links or water pipes or electrical cables, whatever it may be that is being transported. An interesting question is: how much stuff can be carried by this network, and what flow achieves this?

* * *

Consider a directed graph. Let each edge have a label $c(u \to v) > 0$ called the *capacity*. Let there be a *source vertex* s, and a *sink vertex* t. A *flow* is a set of edge labels $f(u \to v)$ such that

It's easy to generalise to multiple sources and sinks, rather harder to generalise to multiple types of stuff.

$$0 \le f(u \to v) \le c(u \to v) \quad \text{on every edge}$$

and

$$\sum_{u:\ u \to v} f(u \to v) = \sum_{w:\ v \to w} f(v \to w) \quad \text{at all vertices } v \in V \setminus \{s, t\}.$$

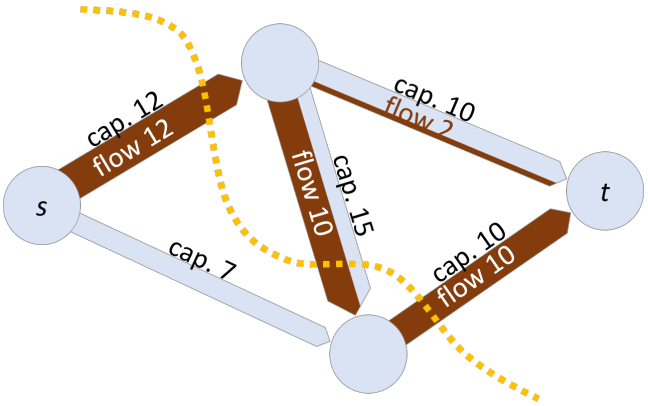
The second equation is called *flow conservation*, and it says that as much stuff comes in as goes out. The *value* of a flow is the net flow out of s,

$$\text{value}(f) = \sum_{u:\ s \to u} f(s \to u) - \sum_{u:\ u \to s} f(u \to s).$$

(It's easy to prove that the net flow out of s must be equal to the net flow into t. See Example Sheet 6.) A *cut* is a partition of the vertices into two sets, $V = S \cup \bar{S}$, with $s \in S$ and $t \in \bar{S}$. The *capacity* of a cut is

$$\text{capacity}(S, \bar{S}) = \sum_{\substack{u \in S,\ v \in \bar{S}: \\ u \to v}} c(u \to v).$$

In this section we will analyse the mathematical properties of flows and cuts. In Section 6.3 we will study an algorithm for computing flows.



A flow of value 12, and a cut of capacity 37

ANALYSIS

Theorem (Max-flow min-cut theorem). *For any flow f and any cut $(S, \bar{S})$,*

$$\text{value}(f) \le \text{capacity}(S, \bar{S}).$$

We could exhaustively enumerate all possible cuts to find the minimum possible value on the right hand side, and this would give an upper bound on the value of any possible flow. Thus,

$$\text{maximum possible flow value} \le \text{minimum cut capacity}.$$

Is it possible to achieve this bound? It is, and the most natural proof is via a flow-finding algorithm, which we'll study in Section 6.3.

Proof. To simplify notation in this proof, we'll extend f and c to all pairs of vertices: if there is no edge $u \to v$, let $f(u \to v) = c(u \to v) = 0$.

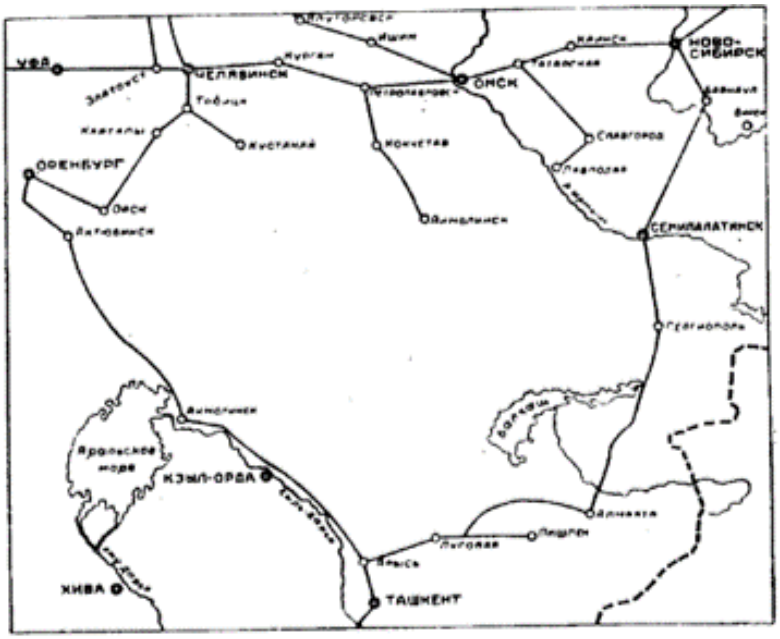
$$\begin{aligned}
\text{value}(f) &= \sum_u f(s \to u) - \sum_u f(u \to s) && \text{by definition of flow value} \\
&= \sum_{v \in S} \left(\sum_u f(v \to u) - \sum_u f(u \to v) \right) && \text{by flow conservation} \\
&&& \text{(the term in brackets is zero for } v \neq s\text{)} \\
&= \sum_{v \in S} \sum_{u \in S} f(v \to u) + \sum_{v \in S} \sum_{u \notin S} f(v \to u) \\
&\quad - \sum_{v \in S} \sum_{u \in S} f(u \to v) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) \\
&&& \text{(splitting the sum over } u \text{ into two sums, } u \in S \text{ and } u \notin S \text{)} \\
&= \sum_{v \in S} \sum_{u \notin S} f(v \to u) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) && \text{by 'telescoping' the sum} \\
&\leq \sum_{v \in S} \sum_{u \notin S} f(v \to u) && \text{since } f \geq 0 \qquad (1) \\
&\leq \sum_{v \in S} \sum_{u \notin S} c(v \to u) && \text{since } f \leq c \qquad (2) \\
&= \text{capacity}(S, \bar{S}) && \text{by definition of cut capacity.}
\end{aligned}$$

This completes the proof. □

APPLICATION

We've already seen how a matching problem can be turned into a flow problem (and then solved!) Now here is a pair of flow problems[10] that inspired the algorithm we'll describe shortly.

The Russian applied mathematician A.N.Tolstoy was the first to formalize the flow problem. He was interested in the problem of shipping cement, salt, etc. over the rail network. Formally, he posed the problem "Given a graph with edge capacities, and a list of source vertices and their supply capacities, and a list of destination vertices and their demands, find a flow that meets the demands."

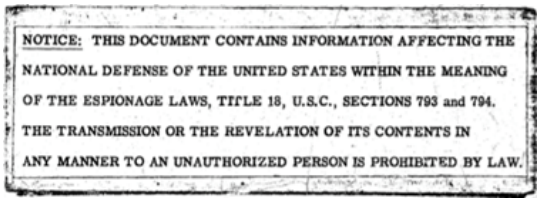


From *Methods of finding the minimum totoal kilometrage in cargo-transportation planning in space*, A.N.Tolstoy, 1930.

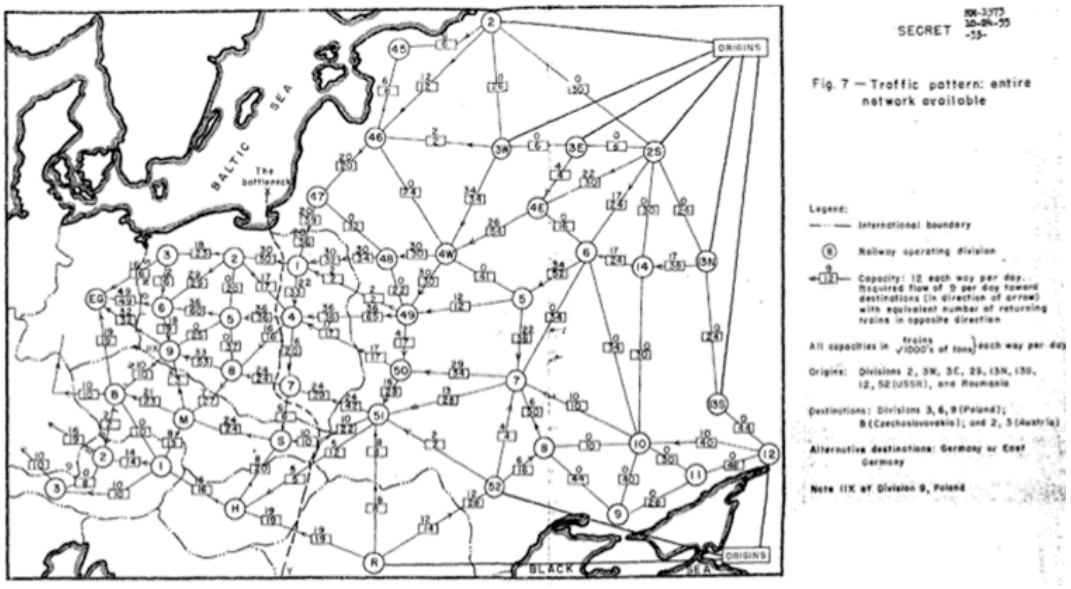
In this illustration, the circles mark sources and sinks for cargo, from Omsk in the north to Tashkent in the south.

[10] For further reading, see *On the history of the transportation and maximum flow problems* by Alexander Schrijver, `http://homepages.cwi.nl/~lex/files/histtrpclean.pdf`; and *Flows in railway optimization* by the same author, `http://homepages.cwi.nl/~lex/files/flows_in_ro.pdf`.

The US military was also interested in flow networks during the cold war. If the Soviets were to attempt a land invasion of Western Europe through East Germany, they'd need to transport fuel to the front line. Given their rail network, and the locations and capacities of fuel refineries, how much could they transport? More importantly, which rail links should the US Air Force strike and how much would this impair the Soviet transport capacity?


NOTICE: THIS DOCUMENT CONTAINS INFORMATION AFFECTING THE NATIONAL DEFENSE OF THE UNITED STATES WITHIN THE MEANING OF THE ESPIONAGE LAWS, TITLE 18, U.S.C., SECTIONS 793 and 794. THE TRANSMISSION OR THE REVELATION OF ITS CONTENTS IN ANY MANNER TO AN UNAUTHORIZED PERSON IS PROHIBITED BY LAW.

From *Fundamentals of a method for evaluating rail net capacities*, T.E. Harris and F.S. Ross, 1955, a report by the RAND Corporation for the US Air Force (declassified by the Pentagon in 1999).



6.3. Ford-Fulkerson algorithm

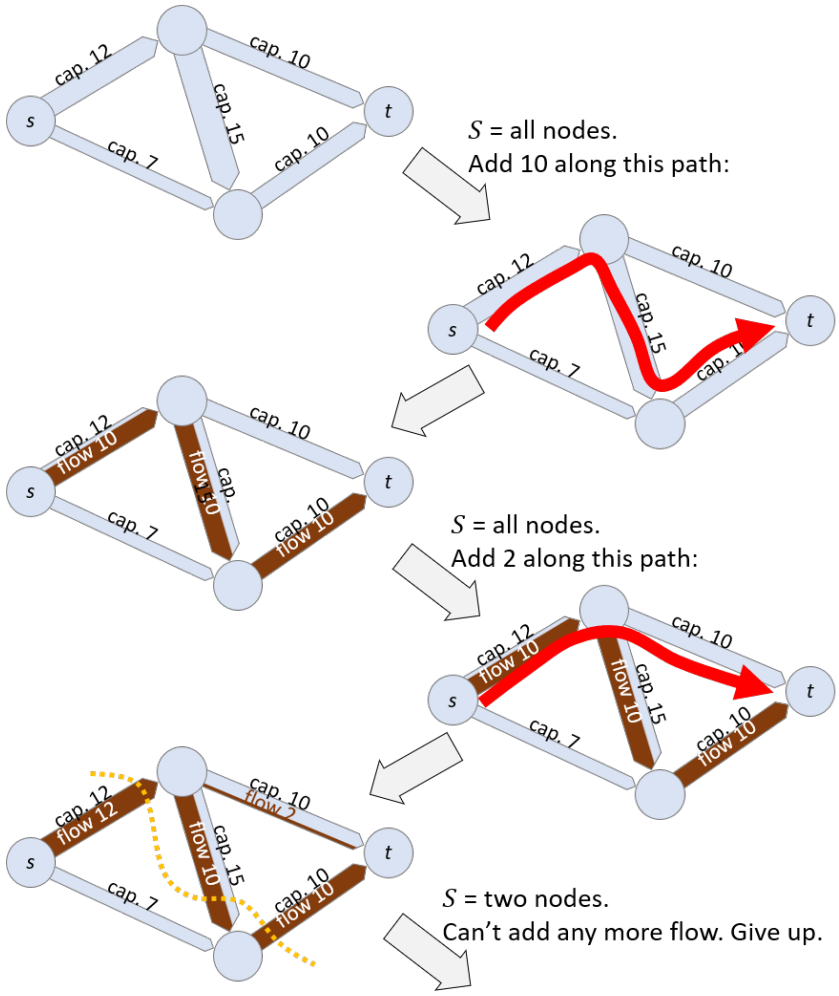
PROBLEM STATEMENT

Given a weighted directed graph g with a source s and a sink t, find a flow from s to t with maximum value (also called a *maximum flow*).

GENERAL IDEA

An obvious place to start is with a simple greedy algorithm: keep on pushing as much flow as we can, starting from s and going to its neighbours, then their neighbours, and so on.

```
1  start with flow = 0
2  while True:
3      let S = {s}
4      # build up S by adding neighbours to which we can push more flow
5      while there is v ∈ S, w ∉ S with f(v → w) < c(v → w):
6          add w to S
7      # add flow if possible, using S as a guide
8      if S includes the sink t:
9          pick any path from s to t in S
10         add as much flow as we can on this path
11         (this path is called the 'augmenting path')
12     else:
13         break
```

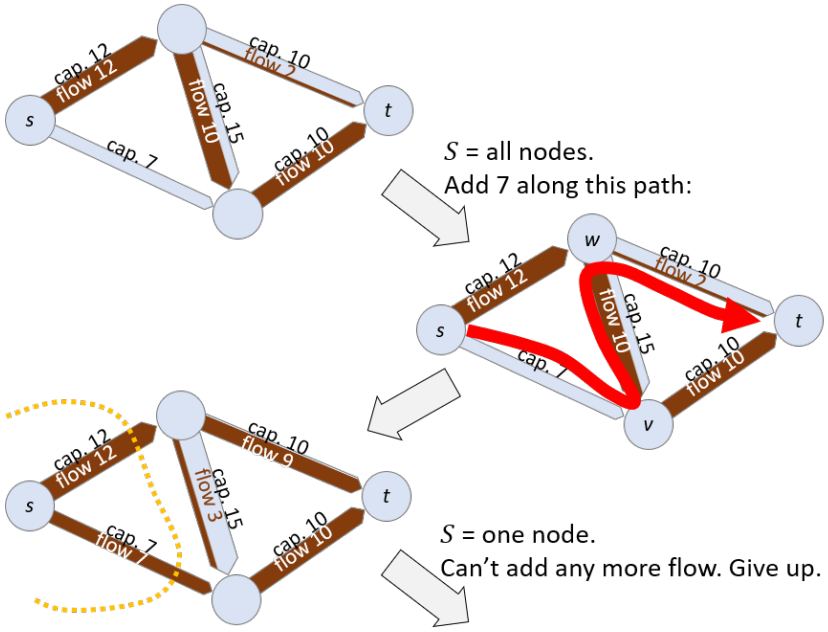


This greedy algorithm found a flow of size 12, then it finished. But we can easily see a flow of size 17 (10 on the top path, 7 on the bottom path). It turns out there is a very simple modification to line 5, to allow the algorithm to reassign flows to 'undo' a mistake.

```
4       # build up S by adding neighbours to which we can send more flow
5       while there is v ∈ S, w ∉ S with f(v → w) < c(v → w) or f(w → v) > 0:
6           add w to S
```

Here's how the algorithm proceeds, with this modification. It first adds v to S, since the edge $s \to v$ is under capacity. Line 5 then sees that $f(w \to v) > 0$, and so it adds w to S, meaning 'I could *reduce* the flow $w \to v$, so in effect I can push more flow to w.' The augmenting path now includes an 'antisense' edge, going against the flow, and line 10 has to be interpreted carefully: we should decrease the flow on that edge.

It's important to understand why this is a valid step. Mathematically speaking, we need to prove that after adding flow on the augmenting path, we still have a valid flow i.e. a flow that satisfies the flow conservation equation. To help you build your intuition, Example Sheet 6 asks you to prove it.



In this example network, our clever-greedy algorithm managed to find a maximum flow. We still need to justify why it works for *any* network (if indeed it does).

IMPLEMENTATION

```
1  def ford_fulkerson(g, s, t):
2      # let f be a flow, initially empty
3      for u→v in g.edges:
4          f(u→v) = 0
5      # Repeatedly find an augmenting path and add flow to it
6      while True:
7          S = Set([s])  # the set of vertices to which we can increase flow
8          while there are vertices v∈S, w∉S with f(v→w)<c(v→w) or f(w→v)>0:
9              S.add(w)
10         if t in S:
11             pick any path p from s to t made up of pairs (v,w) from line 7
12             write p as s = v_0, v_1, v_2, ..., v_k = t
13             δ = ∞ # amount by which we'll augment the flow
14             for each edge (v_i,v_{i+1}) along p:
15                 if v_i→v_{i+1} is an edge of g:
16                     δ = min(c(v_i→v_{i+1}) − f(v_i→v_{i+1}), δ)
17                 else v_i←v_{i+1} must be an edge of g:
18                     δ = min(f(v_{i+1}→v_i), δ)
19             # assert: δ > 0
20             for each edge (v_i,v_{i+1}) along p:
21                 if v_i→v_{i+1} is an edge of g:
```

```
22                        f(v_i → v_{i+1}) = f(v_i→v_{i+1}) + δ
23                    else v_i←v_{i+1} must be an edge of g:
24                        f(v_{i+1}→v_i) = f(v_{i+1}→v_i) − δ
25                # assert: f is still a flow (according to defn. in Section 6.2)
26            else:
27                break  # finished — can't add any more flow
```

This pseudocode doesn't tell us how to choose the path in line 11. One sensible idea is 'pick the shortest path', and this version is called the Edmonds–Karp algorithm. Another sensible idea is 'pick the path that makes δ as large as possible', also due to Edmonds and Karp.

ANALYSIS OF RUNNING TIME

Be scared of the while loop in line 6: how can we be sure it will terminate? In fact, there are simple graphs with irrational capacities where the algorithm does *not* terminate. On the other hand,

Lemma. *If all capacities are integers then the algorithm terminates, and the resulting flow on each edge is an integer.*

Proof. Initially, the flow on each edge is 0, i.e. integer. At each execution of lines 13–18, we start with integer capacities and integer flow sizes, so we obtain δ an integer ≥ 0. It's not hard to prove the assertion on line 19, i.e. that $\delta > 0$. Therefore the total flow has increased by an integer after lines 20–24. The value of the flow can never exceed the sum of all capacities, so the algorithm must terminate. □

Now let's analyse running time, under the assumption that capacities are integer. We execute the while loop at most f^* times, where f^* is the value of maximum flow. We can build the set S and find a path using breadth first search or depth first search, so lines 7–11 can be accomplished in running time $O(V+E)$. Lines 13–24 involve some operations per edge of the augmenting path, which is $O(V)$ since the path is of length $\leq V$. Thus the total running time is $O\big((E+V)f^*\big)$. There's no point including the vertices that can't be reached from s, so we might as well assume that all vertices can be reached from s, so $E \geq V-1$ and the running time can be written $O(Ef^*)$.

It is unsatisfactory that the running time we found depends on the values in the input data (via f^*) rather than just the size of the data. This is unfortunately a common feature of many optimization algorithms, and of machine learning algorithms.

The Edmonds–Karp version of the algorithm can be shown to have running time $O(E^2V)$.

ANALYSIS OF CORRECTNESS

The assertion on line 25, namely that the algorithm does indeed produce a flow, is an exercise on Example Sheet 6. Does it produce a maximum flow?

Theorem. *If the algorithm terminates, and f^* is the final flow it produces, then*

1. *the value of f^* is equal to the capacity of the cut found in lines 7–9;*
2. *f^* is a maximum flow.*

Proof (of 1). Let $(S, \bar{S})$ be the cut. By the condition on line 8, $f^*(w \to v) = 0$ for all $v \in S$, $w \notin S$, so inequality (1) on page 36 is an equality. By the same condition, $f^*(v \to w) = c(v \to w)$ for all $v \in S$, $w \notin S$, so inequality (2) is also an equality. Thus, value(f^*) is equal to capacity$(S, \bar{S})$.

Proof (of 2). Recall the Max-Flow Min-Cut theorem from Section 6.2. It says that for *any* flow f and any cut $(S, \bar{S})$,

$$\text{value}(f) \leq \text{capacity}(S, \bar{S}).$$

Therefore

$$\max_{\text{all flows } f} \text{value}(f) \leq \text{capacity}(S, \bar{S}).$$

But by part 1 we have a flow f^* with value equal to this capacity. Therefore f^* is a maximum flow. □

A cut corresponding to a maximum flow is called a *bottleneck cut*. (The bottleneck cut might not be unique, and the maximum flow might not be unique either, but the maximum flow *value* and bottleneck cut *capacity* are unique.) The RAND report shows a bottleneck cut, and suggests it's the natural target for an air strike.