IA Algorithms 2

Damon Wischik, Computer Laboratory, Cambridge University. Lent Term 2022

Contents

5. Graphs and path finding

A great many algorithmic problems 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, a mathematician from Königsberg, was asked the question "Can I go for a stroll around the city on a route that crosses each bridge exactly once?" He was intrigued – "This question is so banal, but seemed to me worthy of attention in that [neither] geometry, nor algebra, nor even the art of counting was sufficient to solve it." In 1735 he proved the answer was 'No'. His innovation was to turn the question into what we would now call a discrete maths question about a graph, in this case a simple graph with 4 vertices and 7 edges (and he also came up with a clever proof).[1]
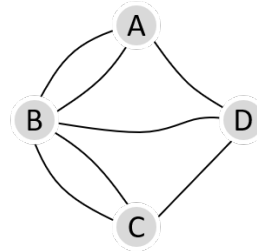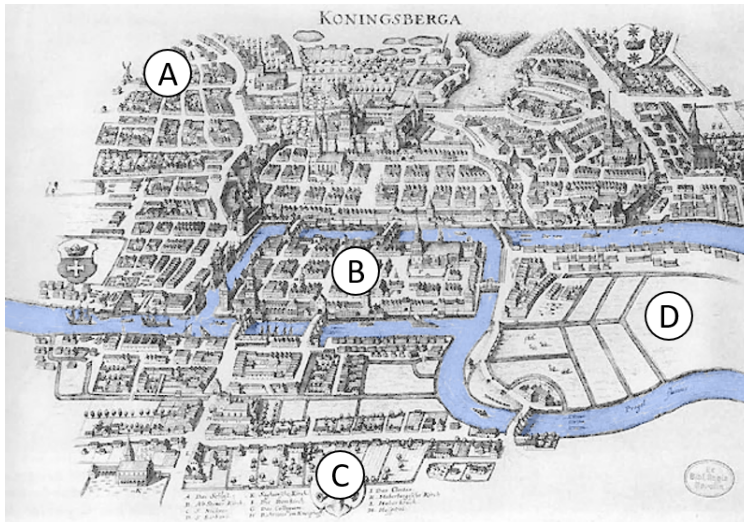
Example. Agents can find paths through a game-map using a graph. Beforehand, (1) draw polygons around the obstacles, (2) subdivide the map into a mesh of convex polygons, (3) define a graph with one vertex per polygon, and edges between adjacent polygons. Then, we can find the agent's shortest path on this graph, using the algorithms we'll study in this course. Once we've found this optimal sequence of polygons, we can go on to find an aesthetically pleasing physical path that walks through them.
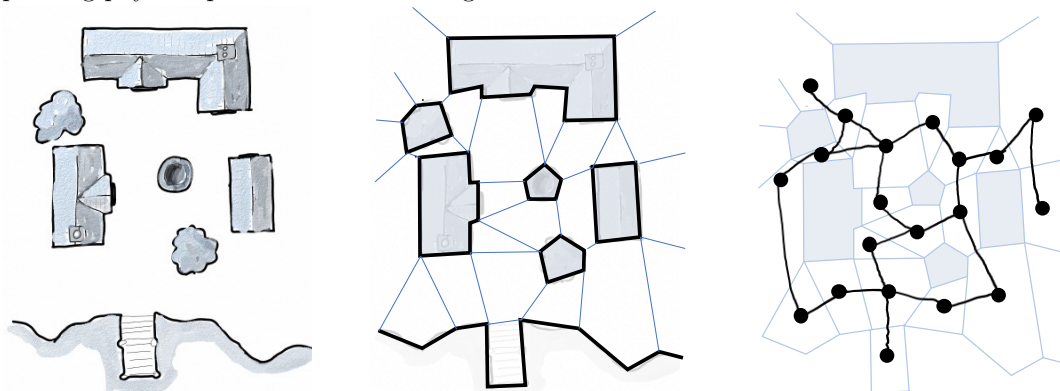
Example. Facebook's underlying data structure is a graph. Vertices are used to represent

---

[1] Euler's proof: Consider any stroll and list the edges it crosses, then count up the number of times each vertex appears. For example, in the stroll [$B \leftrightarrow A$, $A \leftrightarrow B$, $B \leftrightarrow D$, $D \leftrightarrow C$], A appears twice, B appears three times, and so on. Clearly every vertex must appear an even number of times except possibly for the start and end vertices. Now, if there were a stroll that crossed each bridge exactly once, then (by looking at the graph) there are 3 edges at vertex A so A would have to appear 3 times, B would have to appear 5 times, and C and D would have to appear 3 times. But we've already shown that in any stroll there can be at most two vertices that appear an odd number of times, hence no such stroll exists.

users, locations, comments, check-ins, etc. From a Facebook engineering blog post,[2]
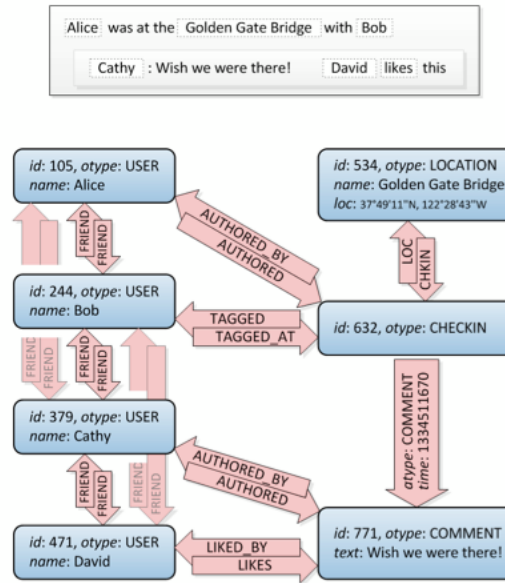
Example. OpenStreetMap represents its map as XML, with nodes and ways. In some parts of the city, this data is very fine-grained. The more vertices and edges there are, the more space it takes to store the data, and the slower the algorithms run. Later in this course we will discuss geometric algorithms which could be used to simplify the graph while keeping its basic shape.


```xml
<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>
```

Exercise. Why do you think Facebook chose to make CHECKIN a type of vertex, rather than an edge from a USER to a LOCATION?

In graphs, edges are only allowed to connect vertices to vertices, they're not allowed to connect vertices to other edges. If we want to be able to attach a COMMENT to a CHECKIN, then CHECKIN has to be a vertex.

---

[2] https://engineering.fb.com/2013/06/25/core-data/tao-the-power-of-the-graph/

□

5.1. Notation and representation

Some notation for describing graphs. A *graph* is a collection of vertices, with edges between them. Write a graph as $g = (V, E)$ where V is the set of vertices and E the set of edges.

- A graph may be *directed* or *undirected*;
  in a directed graph,   an edge from v_1 to v_2 is written $v_1 \to v_2$
  in an undirected graph,   an edge between v_1 and v_2 is written $v_1 \leftrightarrow v_2$.

- The *neighbours* of a vertex v are the vertices you reach by following an edge from v;
  in a directed graph, $\text{neighbours}(v) = \{w \in V : v \to w\}$
  in an undirected graph, $\text{neighbours}(v) = \{w \in V : v \leftrightarrow w\}$.

- A *path* is a sequence of two or more vertices connected by edges;
  in a directed graph, $v_1 \to v_2 \to \cdots \to v_k$
  in an undirected graph, $v_1 \leftrightarrow v_2 \leftrightarrow \cdots \leftrightarrow v_k$.

- A *cycle* is a path from a vertex back to itself, and with no vertices repeated other than $v_1 = v_k$.

In this course, paths are allowed to visit the same vertex more than once. Some people define paths to disallow repeat visits.

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 6.7.

- 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 within a graph in Sections 6.5–6.6.

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.

DAG   forest   forest & tree
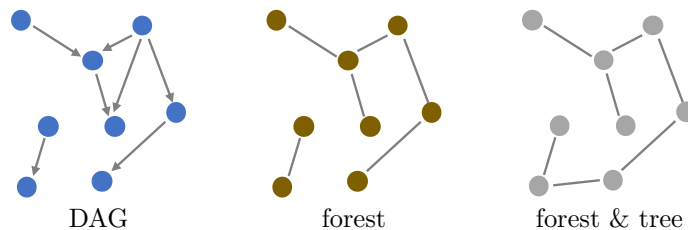
REPRESENTATION

Here are two standard ways to store graphs in computer code: as an array of *adjacency lists*, or as an *adjacency matrix*.

array of
adjacency lists:

adjacency matrix:

	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
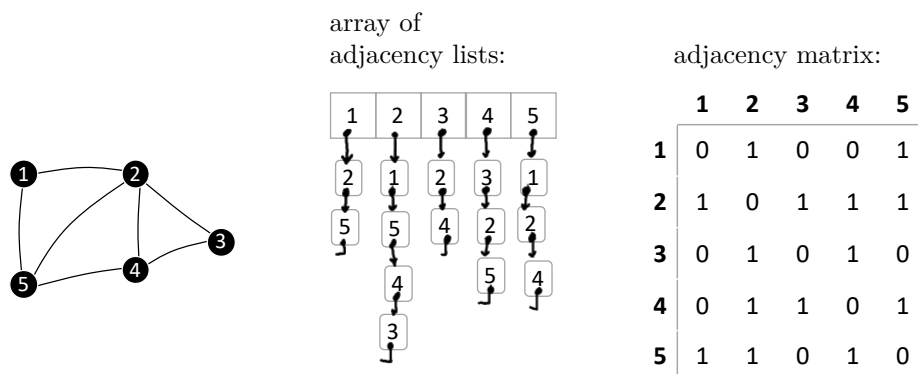
The 'adjacency matrix' representation is good if we need to be able to quickly look up for a given pair of vertices whether there is an edge between them. The 'adjacency list' representation is good if the graph is *sparse*, i.e. if there are many vertices and few edges per vertex.

  What exactly do we mean by a 'good representation'? As usual, we're interested in execution time and in storage requirements, and we'll analyse them using big-O notation so we don't get bogged down in details. For a graph $g = (V, E)$, the storage requirements are

$$\begin{aligned}
\text{adjacency list representation:} &\quad O(V + E) \\
\text{adjacency matrix representation:} &\quad O(V^2)
\end{aligned}$$

(Note: V and E are sets, so we should really write $O(|V| + |E|)$ etc., but it's conventional to drop the $|\cdot|$.) If the graph is sparse, i.e. if $E = o(V^2)$, then the adjacency list doesn't need as much storage. As for algorithm execution time, we'll study that in the following sections.

Exercise.
  (a) What is the largest possible number of edges in an undirected graph with V vertices?
  (b) and in a directed graph?
  (c) What is the smallest possible number of edges in a tree?
  (d) Suppose our graph has multiple edges between a pair of nodes, and that we want to store a label for each edge. If we use an adjacency matrix we'll therefore need to store a list of edges in each cell, not just a number. What is the storage requirement?
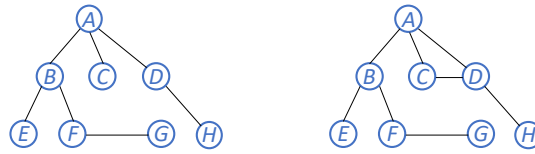

(a) $V(V-1)/2$
(b) $V(V-1)$
(c) $V - 1$
(d) $O(V^2 + E)$

☐

5.2. Depth-first search

A common task is traversing a graph and doing some work at each vertex. For example, a web crawler for a search engine visits every page it can find, and at each page the work is to process the text to add to its search index, then extract all the links in order to find new pages.

GENERAL IDEA: LIKE TRAVERSING A TREE

It's easy to explore a tree, using recursion. If we call visit_tree(D, None) on the graph on the left, which is a tree, then we'll see it visit D, H, A, C, B, E, F, G (or perhaps some other order depending on the order of each vertex's neighbours).

We defined *tree* to mean 'undirected connected acyclic graph'. The tree is drawn as if to suggest that A is the root, but because the graph is undirected there is actually no distinguished vertex, and we're entitled to start the traversal at D.

```
1   # Visit all vertices in the subtree rooted at v
2   def visit_tree(v, v_parent):
3       print("visiting", v, "from", v_parent)
4       for w in v.neighbours:
5           # Only visit v's children, not its parent
6           if w != v_parent:
7               visit_tree(w, v)
```

But if the graph has cycles then this algorithm will get stuck in an infinite recursion. If we run it on the right hand graph above, we might get $D, H, C, A, D, H, C, A, D, ...$ We need some way to prevent this.

DFS IMPLEMENTATION 1: USING RECURSION

Depth-first search uses the same idea as traversing a tree. But we'll explicitly mark which vertices we've already visited, so that we know not to return to them.

```
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)
```

It's always a good idea, when you see a new algorithm, to run through it by hand step-by-step, noting all the decisions it takes. This is especially helpful for understanding recursive algorithms.

Here's the beginning of an execution trace, running on the right hand graph and starting from D. The vertical lines indicate which function call we're currently inside.

```
dfs_recurse(g, D):
|  visit(D):
|  |  neighbours = [H,C,A]
|  |  visit(H):
|  |  |  neighbours = [D]
|  |  |  don't visit D
|  |  |  return from visit(H)
|  |  visit(C):
|  |  |  neighbours = [D,A]
|  |  |  don't visit D
|  |  |  visit(A):
|  |  |  |  ...
```

DFS IMPLEMENTATION 2: WITH A STACK

Now a second implementation, in which we'll keep a record of which vertices are waiting to be explored, and jump straight to the next one.
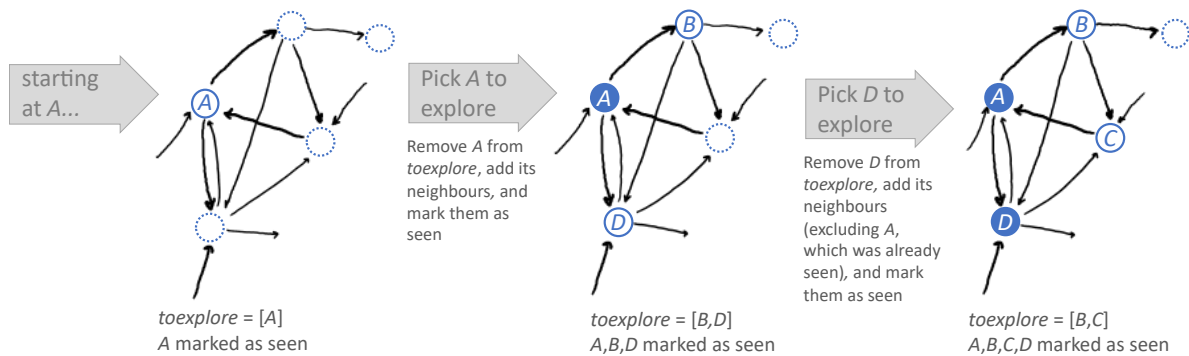

dfs_recurse   *dfs* using a Stack
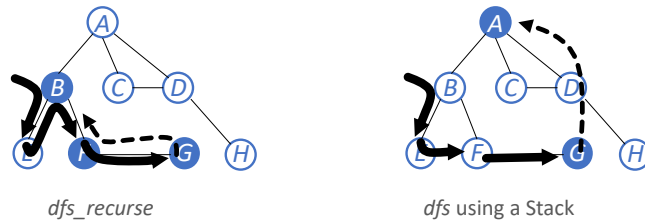
To see why this is different, consider the execution trace of **dfs_recurse** starting at B:

```
visit(B):
  visit(E) from B, return from E to B
  visit(F) from B
    visit(G) from F, return from G to F,
    return from F to B
  visit(A) from B
```

Instead of this chain of **return**s, could we jump straight from G to A, the next vertex waiting to be explored? Obviously we'd need to know that A is waiting to be explored, so we'd need to have noted this down when we first visited B. We can use a Stack, a Last-In-First-Out data structure, to store the list of vertices waiting to be explored; this way we'll visit children before returning to distant cousins.

There is a subtle difference between **dfs** and **dfs_recurse** as implemented here — they don't actually visit vertices in the same order as each other. The example sheet asks you to look into this, and to modify **dfs** so that they do.

```
1   # Visit all vertices reachable from s
2   def dfs(g, s):
3       for v in g.vertices:
4           v.seen = False
5       toexplore = Stack([s])    # a Stack 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.pushright(w)
13                  w.seen = True
```


starting at A...

toexplore = [A]
A marked as seen

Pick A to explore

Remove A from *toexplore*, add its neighbours, and mark them as seen

toexplore = [B,D]
A,B,D marked as seen

Pick D to explore

Remove D from *toexplore*, add its neighbours (excluding A, which was already seen), and mark them as seen

toexplore = [B,C]
A,B,C,D marked as seen

ANALYSIS

In **dfs**, (a) line 4 is run for every vertex which takes $O(V)$; (b) lines 8–9 are run at most once per vertex, since the **seen** flag ensures that each vertex enters **toexplore** at most once, so the running time is $O(V)$; (c) lines 10 and below are run for every edge out of every vertex that is visited, which takes $O(E)$. Thus the total running time is $O(V + E)$.

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) lines 9 and below are run for every edge out of every vertex visited.

* * *

Pay close attention to the clever trick in analysing the running time. We didn't try to build up some complicated recursive formula about the running time of each call to visit, or to reason about when which part of the graph was put on the stack. Instead we used mathematical reasoning to bound the total number of times that a vertex could possibly be processed during the entire execution. This is called *aggregate analysis*, and we'll see more examples later in the course when we look at the design of advanced data structures.

The recursive implementation uses the language's call stack, rather than our own data structure. Recursive algorithms are sometimes easier to reason about, and we'll use the recursive implementation as part of a proof in Section 6.7.

5.3. Breadth-first search

A common task is finding paths in a graph. With a tiny tweak to the depth-first search algorithm, we can find shortest paths.

GENERAL IDEA

Suppose we want to find the shortest path from A to some other vertex, in a directed graph. Let's start by rearranging the graph, to put A at the top, then all the vertices at distance 1 underneath, and all the nodes at distance 2 underneath that, and so on. (By 'distance d' we mean 'can be reached by a path with d edges, and cannot be reached by a path with $< d$ edges'.) The graph rearranges itself into two parts: there is a tree consisting of all the vertices, with edges that go down by exactly one 'level'; and there are extra edges that go either horizontally or up.
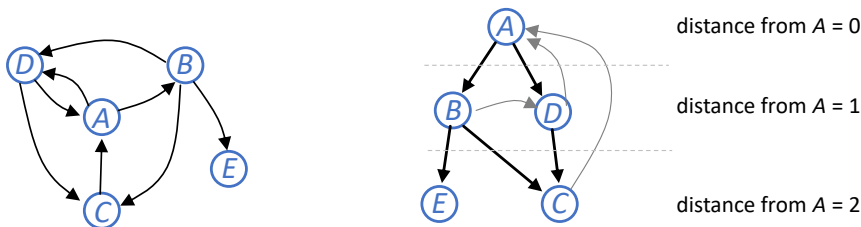
The idea of breadth first search is to visit A, then all nodes at distance 1 from A, then all nodes at distance 2, and so on. In other words, we'll explore the breadth of the tree first, before going deeper. There's a very simple way to achieve this. Suppose we've already visited all the vertices at level $< d$, and we've got a list of all the vertices at level d: then we just go through that list and add all the vertices that are newly reachable; these must be vertices at level $d + 1$. Keep going until there's nothing more to visit.
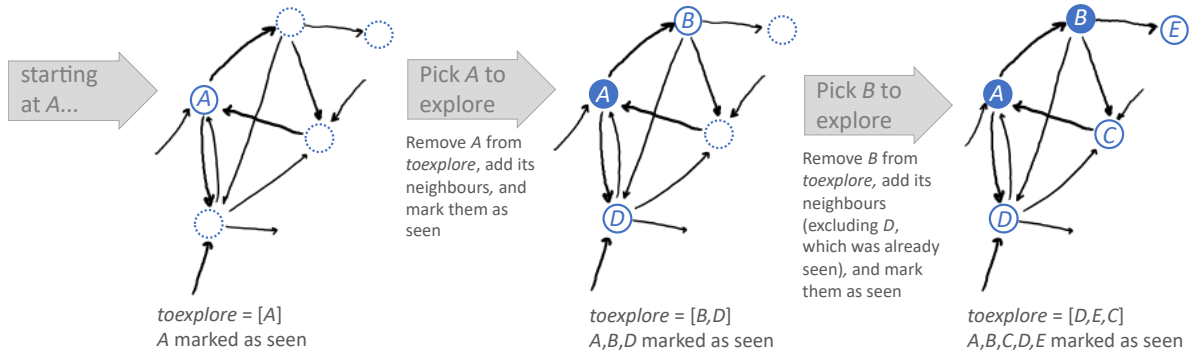
IMPLEMENTATION

To implement the breadth-first strategy, we don't even need to manage 'list of vertices at distance d'. All we need is a Queue to store all the vertices we're waiting to explore. Push new vertices on the right of the queue, pop vertices from the left, and that way we're guaranteed to pop all vertices in correct order of distance.

The code turns out to be almost identical to dfs. The only difference is that it uses a Queue instead of a Stack.

```
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.popleft()   # Now visiting vertex v
10          for w in v.neighbours:
11              if not w.seen:
12                  toexplore.pushright(w)
13                  w.seen = True
```

starting at A...

toexplore = [A]
A marked as seen

Pick *A* to explore

Remove *A* from *toexplore*, add its neighbours, and mark them as seen

toexplore = [B,D]
A,B,D marked as seen

Pick *B* to explore

Remove *B* from *toexplore*, add its neighbours (excluding *D*, which was already seen), and mark them as seen

toexplore = [D,E,C]
A,B,C,D,E marked as seen

We can adapt this code to find a path between a pair of nodes: we just need to keep track of how we discovered each vertex. For every vertex at distance d, we'll store a come_from arrow, pointing to a vertex at distance $d - 1$. Here's a picture, then the code.

Starting at A...

Pick *A* to explore

Pick *B* to explore
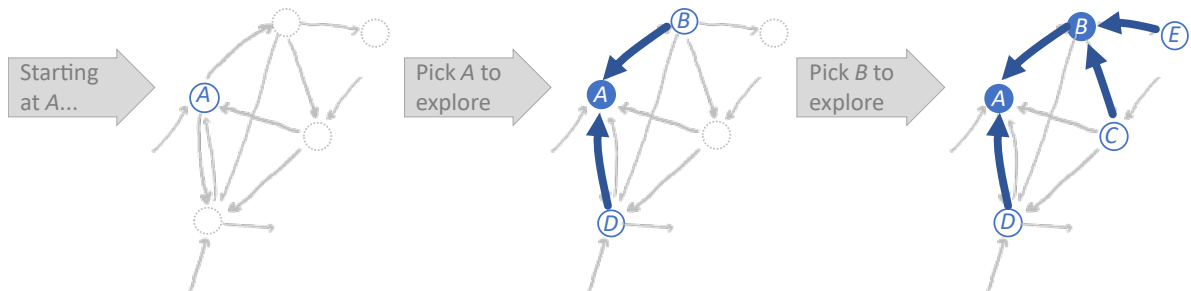
```
1   # Find a path from s to t, if one exists
2   def bfs_path(g, s, t):
3       for v in g.vertices:
4           v.seen = False
5           v.come_from = None
6       s.seen = True
7       toexplore = Queue([s])
8
9       # Traverse the graph, visiting everything reachable from s
10      while not toexplore.is_empty():
11          v = toexplore.popleft()
12          for w in v.neighbours:
13              if not w.seen:
14                  toexplore.pushright(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

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

* * *

Here is another way to think of 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, via the Queue, without needing an explicit variable to store d.
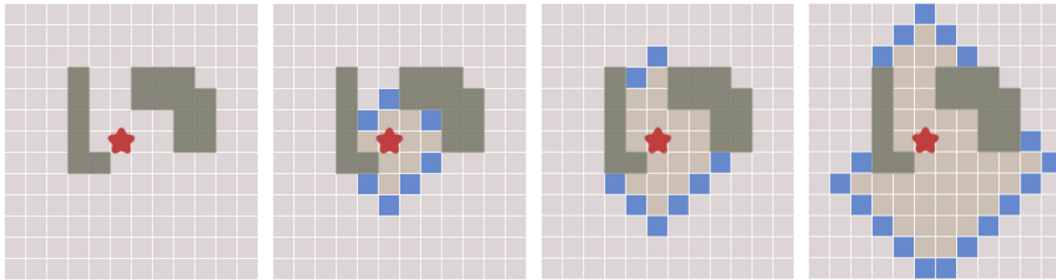
In this illustration[3], we're running bfs starting from the blob in the middle. The graph has one vertex for each light grey grid cell, and edges between adjacent cells, and the black cells in the left hand panel are impassable. The next three panels show some stages in the expanding frontier.

---

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

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 with its travel time: how do we find the quickest route between two locations?

Let's express the problem more precisely. Let $\text{cost}(u \to v)$ be the cost associated with edge $u \to v$, let the cost of a path be the sum of its edge costs, and define $\text{distance}(u \text{ to } v)$ to the the minimum cost of all paths from u to v. If there is no path from u to v, let $\text{distance}(u \text{ to } v) = \infty$. Any path which achieves the minimum cost is called a *shortest path*; as with breadth-first search, once we can work out distances, it's easy to find shortest paths.

Here's an illustration[4]. 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.
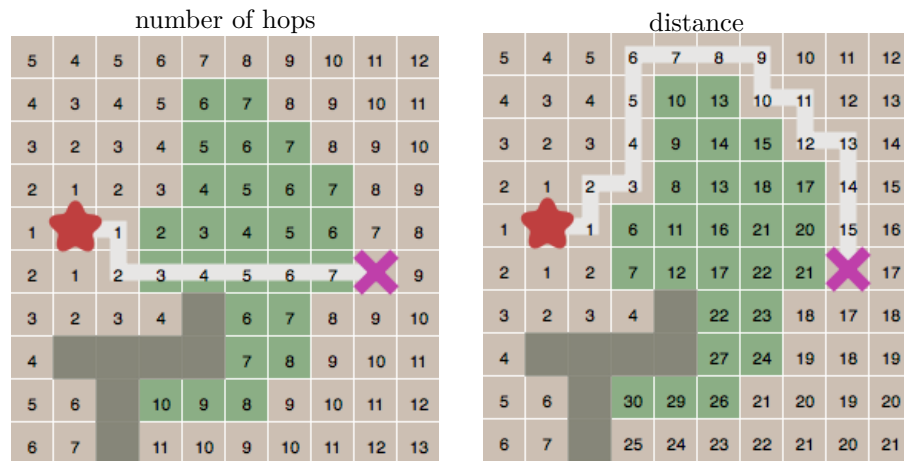
PROBLEM STATEMENT

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.

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.

---

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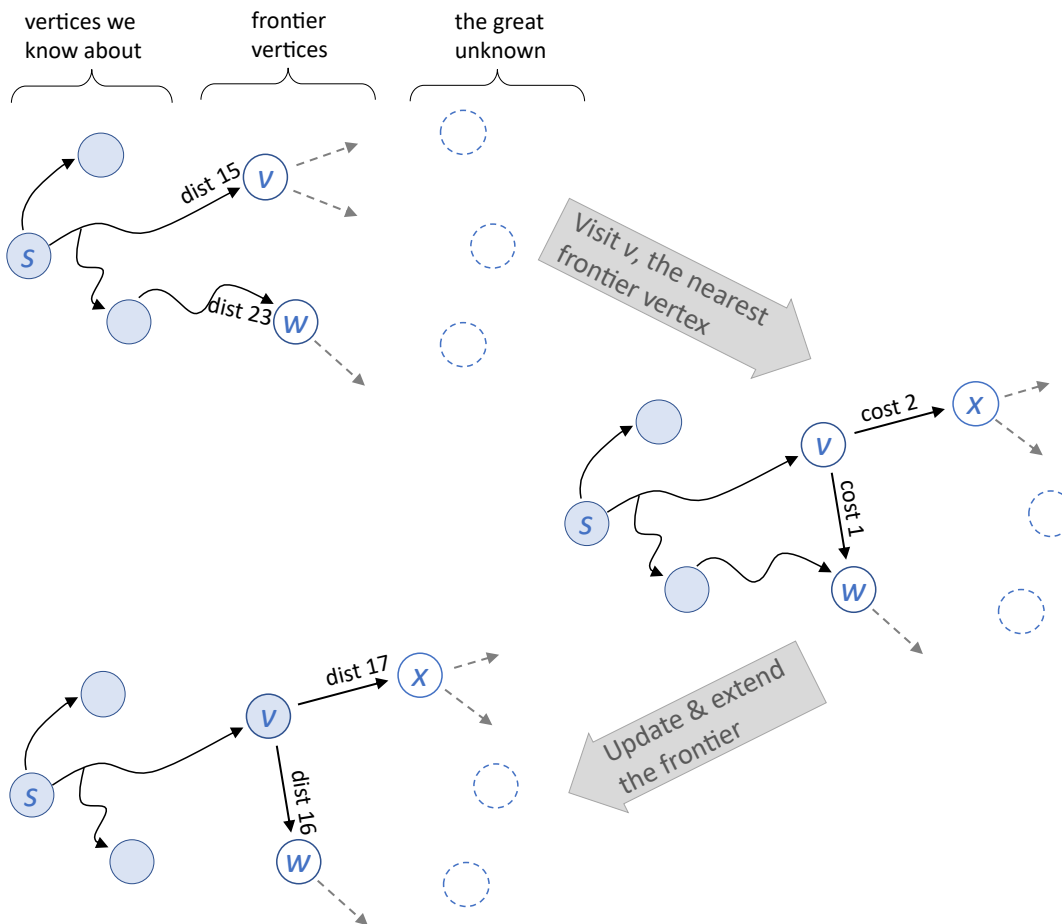
IMPLEMENTATION

This algorithm was invented in 1959 and is due to Dijkstra[5] (1930–2002), a 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 4.8 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
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

---

[5]Dijkstra was 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."

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.   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 the total running time is

$$O(V) + O(V) \times \text{cost}_{\text{popmin}} + O(E) \times \text{cost}_{\text{push/dec.key}}$$

where the individual operation costs depend 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(). Since the number of items stored in the heap at any time is $\leq V$, by the assertion on line 10, the total running time is $O(E + V \log V)$.

In this theorem, v.distance is a variable that is updated during program execution, and distance$(s \rightarrow v)$ is the Platonic mathematical object. 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.

Theorem (Correctness).   *The dijkstra algorithm terminates. When it does, for every vertex v, the value v.distance it has computed is equal to* distance$(s$ *to* $v)$. *Furthermore, the two assertions never fail.*

Proof (that Assertion 9 never fails). Suppose this assertion fails at some point in execution. Let v be the vertex for which it first fails, and let T be the time of this failure. Consider a shortest path from s to v. (This means the Platonic mathematical object, *not* a computed variable.) Write this path as

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

There are two cases to consider: CASE1 in which one of these vertices hasn't yet been popped from toexplore by time T, and CASE2 in which they have all been popped.

Consider CASE1 first, and let i be the index of the first vertex in the sequence that, at time T, hasn't been popped. So the path is

$$s = \underbrace{u_1 \rightarrow u_2 \rightarrow \cdots \rightarrow u_{i-1}}_{\text{already popped}} \rightarrow \underbrace{u_i}_{\substack{\text{not yet} \\ \text{popped}}} \rightarrow \underbrace{\cdots \rightarrow u_k}_{\text{???}} = v$$

(We've just popped $v = u_k$, so we know $i < k$. The vertices between i and k, if there are any, might or might not have been popped by time T.) Now, reasoning about the stored .distance variables as they stand at time T,

distance$(s$ to $v)$

$< v$.distance   since the assertion failed at v

$\leq u_i$.distance   (*)

$\leq u_{i-1}$.distance $+ \text{cost}(u_{i-1} \rightarrow u_i)$ by lines 13–18 when u_{i-1} was popped

$= \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \rightarrow u_i)$ assertion didn't fail at u_{i-1}

$\leq \text{distance}(s \text{ to } v)$ since $s \rightarrow \cdots u_{i-1} \rightarrow u_i$ is on a shortest path s to v.

The tricky step is (*). This line is because we just popped v from the PriorityQueue, and we know u_i was in there also because it would have been forced in when we popped u_{i-1}, and the PriorityQueue gave us v rather than u_i, hence v.distance $\leq u_i$.distance. Thus, we obtain a contradiction.

In CASE2, we also obtain a contradiction, and it's easier to prove:

distance$(s$ to $v)$

$< v$.distance   since the assertion failed at v

$\leq \text{distance}(s \text{ to } u_{k-1}) + \text{cost}(u_{k-1} \rightarrow v)$ by lines 13–18 when u_{k-1} was popped

$= \text{distance}(s \text{ to } v)$ since $s = u_1 \rightarrow \cdots \rightarrow u_k = v$ is a shortest path.

This again is a contradiction.

We have proved that, if Assertion 9 fails at some point in execution, there is a contradiction. Thus, it can never fail.

Proof that Assertion 10 never fails. Once a vertex v has been popped, Assertion 9 guarantees that $v.\mathsf{distance} = \text{distance}(s \text{ to } v)$. 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 $v.\mathsf{distance}$ 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 'BREAKPOINT' PROOF STRATEGY

It's worth stepping back and looking at the proof strategy we used here, since it's one we'll use again and again. I call it the 'breakpoint' strategy.

We decided on a property which we want to be true at all points during execution. We designed this property so that (1) if it's true when the algorithm terminates, then the output of the algorithm is correct, and (2) if it's true up to an arbitrary timepoint $T - 1$, then it must be true at time T. The proof of (2) is rather like setting a breakpoint in a debugger and arguing that, based on the guarantees so far about the algorithm's data structures, what happens in the next few lines of code can't possibly go wrong.

Formally, a breakpoint proof is proof by induction. Any proof by induction requires us to specify an ordering, and here the ordering is by execution time. The induction hypothesis is "our property holds at all instants up to $T - 1$", and the induction step is to prove that it must therefore hold at time T.

For some other graph algorithms, it's helpful use induction but applied to a different order, e.g. by distance rather than execution time. Whenever you use proof by induction, make sure you say explicitly what your ordering is.

5.5. Algorithms and proofs

Here's what Dijkstra had to say about programming and proofs:

> *Right from the beginning, and all through the course, we stress that the program-mer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.*[6]

> *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.*[7]

> *Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.*[8]

This course is 50% about the ideas behind the algorithms, 50% about the proofs. Don't think of proofs as hoops that a cruel lecturer forces you to jump through! If you can't figure out a correct proof that your algorithm works, chances are there's a bug in your algorithm, for example a special case that you haven't coded for. Conversely, every nifty trick that you invented to make your algorithm work is likely to have a counterpart in a proof of correctness, otherwise the trick would be superfluous. Algorithm proofs are just a tool for making sure our code works correctly and for clarifying in our heads the big ideas behind an algorithm.

Here is an exam question about Dijsktra's algorithm, and a selection of mangled proofs. The first thing an examiner checks for is whether the proof passes the 'smell test'. Does this answer have all the key ingredients from Dijkstra's proof—the 'breakpoint' proof strategy, the reliance on edge weights being non-negative? If it fails the smell test, the examiner will look for a faulty inference, in other words construct a counterexample to demonstrate the fault.

Let $\mathsf{dijkstra\_path}(g, s, t)$ be an implementation of Dijkstra's shortest path algorithm that returns the shortest path from vertex s to vertex t in a graph g. Prove that the implementation can safely terminate when it first encounters vertex t.

Bad Answer 1. At the moment when the vertex t is popped from the priority queue, it has to be the vertex in the priority queue with the least distance from s. This means that any other vertex in the priority queue has distance $\geq$ that for t. Since all edge weights in the graph are ≥ 0, any path from s to t via anything still in the priority queue will have distance $\geq$ that of the distance from s to t when t is popped, thus the distance to t is correct when t is popped.

*This fails the smell test in two ways. First, there is no hint of induction—the proof only discusses what happens when the target vertex t is popped. Second, it doesn't distinguish between the two 'distances': (a) v.**distance**, the quantity computed and updated in the course of the algorithm, and (b) distance(s to v), the true mathematical distance. The point of Dijkstra's algorithm is that the former eventually becomes equal to the latter.*

> *Where exactly is the faulty inference? The first sentence is about how priority queues work, so it must be referring to distances as computed by the algorithm, not to true mathematical distances; but the last sentence seems to be referring to true mathematical distances. So let's set up a counterexample where the two are at odds. In this diagram, u and t are in the priority queue, and the nodes have been labelled by their computed distances at the instant when t is popped.*

---

[6]EWD 1036: On the cruelty of really teaching computing science, `https://www.cs.utexas.edu/~EWD/ewd10xx/EWD1036.PDF`

[7]EWD 340: The humble programmer, `https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF`. Quite the opposite of test-driven development! Contrast to Donald Knuth, another pioneer of computer science, who once wrote "Beware of bugs in the above code; I have only proved it correct, not tried it."

[8]EWD 498: How do we tell truths that might hurt? `https://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF`
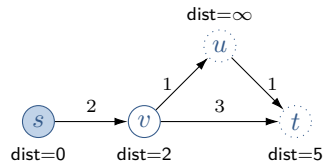
*What does the Bad Answer say about this scenario? We're about to pop t, since t.**distance** < u.**distance**. And yet the true mathematical distance of the path $s \to v \to u \to t$ is shorter than the computed t.**distance**, so the final sentence of the Bad Answer is incorrect.*

One might say "Oh, but the proof in Section 5.4 shows that this can't happen." True … but an answer to the question should PROVE that this scenario can't happen (or make explicit reference to lecture notes!), and because this answer doesn't it's a Bad Answer.

☐

Bad Answer 2. Dijkstra's algorithm performs a breadth-first search on the graph, storing a frontier of all unexplored vertices that are neighbours to explored vertices.

Each time it chooses a new vertex v to explore, from the frontier of unexplored vertices, it chooses the one that will have the shortest distance from the start s, based on the edge weight plus the distance from s of its already explored neighbour.

Given that no other vertex in the frontier is closer to s, and that this new vertex v has yet to be explored, when v is explored it must have been via the shortest path from s to v.

Hence, when t is first encountered, it must have been found via its shortest path and the program can safely terminate.

This answer is slightly better than the previous one because it nods in the direction of induction—it makes an argument about what happens "each time it chooses a new vertex", not just about what happens when it chooses t. But the ordering behind the induction isn't made clear. And it still fails the smell test because it makes no distinction between 'distance as computed' and 'true mathematical distance'.
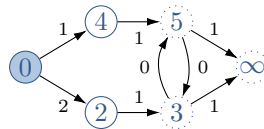

*Let's look for a counterexample. In this graph, each node has been labelled with its current **distance**, and each edge with its cost. Node 0 is the start vertex, node ∞ is the destination vertex, nodes 0, 4, and 2 have already been explored, and nodes 3 and 5 are in the frontier. The algorithm will pick node 3 to explore, since this has the smallest **distance** variable, but this **distance** is incorrect and the corresponding path (via node 2) is wrong.*

The issue is that the algorithm should never get into the state shown here! But the Bad Answer doesn't argue that this state is impossible: instead it just assumes that when v is popped all the already-explored vertices are correct. The proof needs to say 'by induction on vertices in the order they are explored'. Only then is it legitimate to assume that all the already-explored vertices are correct.

☐

> Exercise. Bad Answer 2 doesn't state anywhere that edge weights need to be ≥ 0. Explain which line of the proof fails when the algorithm is run on this graph:
>
>
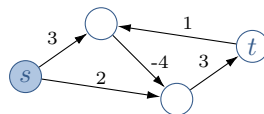
Good Answer. We're asked to show that the implementation can safely terminate 'when it first encounters vertex t'. We'll take this to mean 'when it first pops t', since otherwise the claim is false. (To see it's false, consider a graph with a very-large-weight edge from s to t,

and a short-distance multihop path from s to t. It will encounter t when it's enumerating the neighbours of s, but at that point in the execution it will not have found the true distance to t, nor the true shortest path.)

The proof of correctness from section 5.4 page 14 shows that Assertion 9 never fails. This proof carries through unchanged to the early-terminating version of Dijkstra's algorithm that we're considering here. Hence, at the instant we pop t, t.distance is equal to the true distance from s to t.

The algorithm will only set v.distance on a vertex v once it has found a path from s to v whose cost is equal to v.distance. Therefore, at the instant we pop t, we have a path from s to t of distance t.distance. We have just argued that this is the true distance from s to t, hence by definition of 'distance' we have a shortest path. Therefore, we may safely terminate.

If the question tells us we may quote results from lectures, we don't need to repeat the proof about Assertion 9. If not, we should reproduce the proof in our answer. That proof contains all the key ingredients—the 'breakpoint' proof strategy, and the reliance on non-negative edge weights.

The question doesn't actually tell us how dijkstra_path(g, s, t) is implemented, nor does it ask us to provide an implementation. So we can take it for granted that its mechanism for getting a path is correct, so we don't need to justify any further the claim that when it sets v.distance we have a path of that distance.

□

5.6. Bellman-Ford

Now for a new wrinkle: graphs whose edges can have both positive and negative costs.

We'll use the term *weight* rather than cost of an edge. (The words 'cost' and 'distance' suggests positive numbers, and so they give us bad intuition for graphs with negative costs.) The weight of a path is the sum of its edge weights, and our goal is to find *minimum weight paths* from some start vertex.
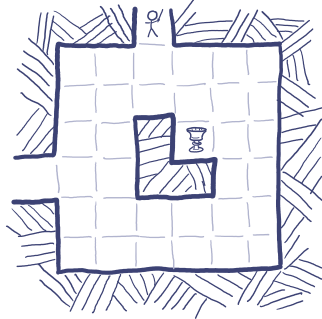

Example (Planning problems). Consider an adventurer who has just entered a room in a dungeon, and wants to quickly get to the other exit, but has the option of detouring to pick up treasure. We could frame this as seeking to minimize

$$W = \begin{cases} T - r & \text{if we pick up treasure} \\ T & \text{otherwise} \end{cases}$$

where T is the time to reach the exit and r is the value of the treasure. What is the optimal path, and does it involve picking up the treasure?

We can turn this into a directed graph problem. Let there be a vertex for every state the game can be in: this comprises both the location of the adventurer, as well as a flag saying whether the treasure is still available. And let there be edges for all the possible game moves. Give each edge weight 1, except for the moves which involve picking up the treasure, which have weight $1 - r$. In graph language, we're told the start vertex, and we have the choice of two possible destination vertices (exit with treasure, and exit without treasure), and we want to find the weights of minimum weight paths to those two destinations.   ◇

Example (Negative cycles). What's the minimum weight from a to b in the graph below? 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
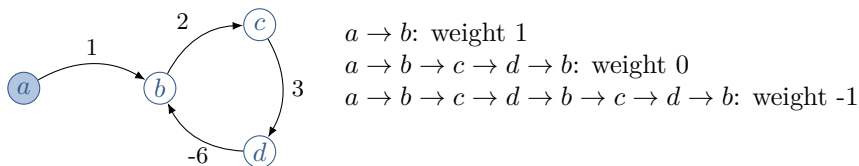
Exercise. Run Dijkstra's algorithm by hand on each of these graphs, starting from the shaded vertex. The labels indicate edge weights. What happens?
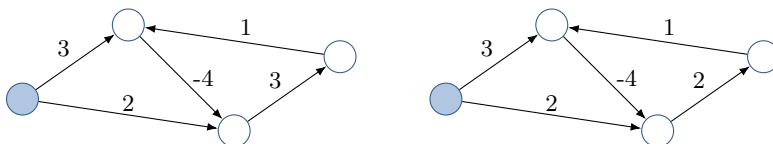
GENERAL IDEA

Dijsktra's algorithm can fail on graphs with negative edge weights. *[Before continuing, do the exercise!]* But the update step at the heart of Dijkstra's algorithm, lines 13–14 on page 13, is still sound. Let's restate it. If we've found a path from s to u, call it $s \rightsquigarrow u$, and if there is an edge $u \to v$, then $s \rightsquigarrow u \to v$ is 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
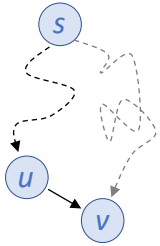
$$\text{if } v.\text{minweight} > u.\text{minweight} + \text{weight}(u \to v):$$
$$\text{let } v.\text{minweight} = u.\text{minweight} + \text{weight}(u \to v)$$

This update rule is known as *relaxing the edge* $u \to v$.

The idea of the Bellman-Ford algorithm is to just keep relaxing all the edges in the graph, over and over again, updating the minweight every time we find a better path. The magic is that we only need to apply it V times.

Dijkstra	Bellman-Ford
can get stuck in an ∞ loop if some weights < 0	always terminates
$O(E + V \log V)$ if all weights ≥ 0	$O(VE)$
visits vertices in a clever order, relaxes each edge once	visits vertices in any order, relaxes each edge multiple 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 minimum-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 minimum-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.

$$v_0.\text{minweight} + \text{weight}(v_0 \to v_1) \geq v_1.\text{minweight}$$
$$v_1.\text{minweight} + \text{weight}(v_1 \to v_2) \geq v_2.\text{minweight}$$
$$\vdots$$
$$v_k.\text{minweight} + \text{weight}(v_k \to v_0) \geq v_0.\text{minweight}$$

Putting all these equations together,

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

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. □

5.7. Dynamic programming

Dynamic programming means figuring out how to express a problem in terms of easier sub-problems. Richard Bellman (1920–1984) invented dynamic programming in order to solve planning problems, which, as we saw in section 5.6, can be thought of as finding minimum weight paths on a graph. Our graph has a vertex for every state that the system can be in, and an edge for every transition. Edges are labelled with rewards. ('Reward' just means 'negative cost'. In graph problems it's common to think in terms of costs and distances, and in planning problems it's more common to think in terms of rewards.)
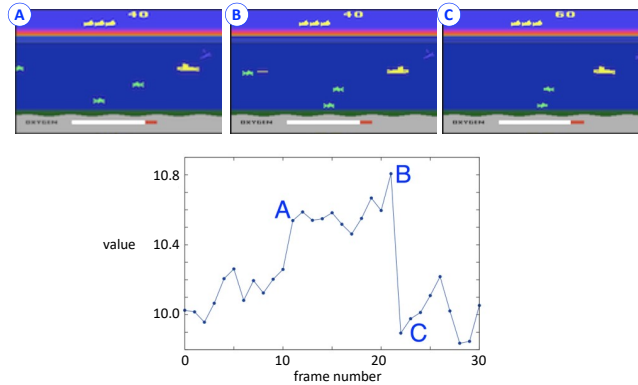
The heart of dynamic programming is a recurrence equation for the *value function*. The value function evaluates, for any state v, the expected future reward that can be gained starting from v. The picture above illustrates 30 frames of gameplay of the Atari game Seaquest, and the value function that was computed by DeepMind.[9] The player is the yellow submarine on the right. At time A a new enemy appears on the left, and the value function increases because of the potential reward. At time B the player's torpedo has nearly reached the enemy, and the value function is very high anticipating the reward. (The value function shown here is based on a reward of $+1$ whenever the player gains points.) At time C the enemy has been hit, and the reward has been won, so the value function reverts to its baseline.

GENERAL IDEA

Define the value function $F_{dst,t}(v)$ to be the minimum weight for reaching vertex *dst* within t timesteps starting from v, assuming that it takes one timestep to follow an edge.
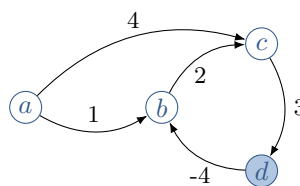
To illustrate, here's the value function for reaching state d, for the simple graph shown above:

$$F_{d,1}(v) = \begin{cases} 3 & \text{if } v = c \text{ with the one-hop path } c \to d \\ 0 & \text{if } v = d \text{ since we're already there} \\ \infty & \text{otherwise, since we can't reach } d \text{ within 1 timestep} \end{cases}$$

$$F_{d,2}(v) = \begin{cases} 5 & \text{if } v = b \text{ by following } b \to c \to d \\ 7 & \text{if } v = a \text{ by following } a \to c \to d \\ 3 & \text{if } v = c \text{ with the single-hop path } c \to d \\ 0 & \text{if } v = d \text{ since we're already there} \end{cases}$$

and so on.

---

[9] *Playing Atari with Deep Reinforcement Learning*, Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller, 2013, `https://arxiv.org/pdf/1312.5602v1.pdf`. This is the paper that kickstarted the deep reinforcement learning revolution. See `https://blog.evjang.com/2018/08/dijkstras.html` for an excellent blog post by Eric Jang, that discusses the link between reinforcement learning and shortest paths.

For a general graph, we can write down a recurrence equation (called the *Bellman equation*) for the value function:

$$F_{d,t}(v) = \min\left(F_{d,t-1}(v), \min_{w:v\to w}\left\{\text{weight}(v \to w) + F_{d,t-1}(w)\right\}\right).$$

In words, "If we have a path $v \rightsquigarrow d$ that takes $\le t - 1$ steps, that's obviously a valid path of $\le t$ steps; alternatively, we could take a first step $v \to w$ and then take the optimal path $w \rightsquigarrow d$ in $\le t - 1$ steps." The terminal condition is

$$F_{d,0}(v) = \begin{cases} 0 & \text{if } v = d \\ \infty & \text{otherwise.} \end{cases}$$

We can simply iterate this equation, in the usual dynamic programming way: first write down $F_{d,0}(v)$ for all v, then compute $F_{d,1}(v)$ for all v, and so on. The value function itself tells us the weight of a minimum weight path, and we can recover the path by reading off which option gives the minimum at each timestep.

PROBLEM STATEMENT

Given a directed graph where each edge is labelled with a weight, and assuming it contains no negative-weight cycles, then for every pair of vertices compute the weight of the minimum-weight path between those vertices.

Why the condition about negative-weight cycles, and why is there no mention of time horizon? The time horizon in the Bellman equation was crucial—it's what lets us break the problem down into easier subproblems. There's a simple reason. If the graph has no negative-weight cycles, then any minimum weight path must have $\le V$ vertices (if it had more then there must be a cycle, which by assumption has weight ≥ 0, so we might as well excise it). Thus for any pair of vertices there is a minimum weight path between them with $\le V - 1$ edges. So what we want to compute is $F_{d,V-1}(v)$.

The problem statement says *for every pair of vertices*. There's a nifty way to compute the value function using matrices, and all-to-all minimum weights just drop out with no extra work. This implementation has running time $O(V^3 \log V)$.

MATRIX IMPLEMENTATION (NON-EXAMINABLE)

Let $M_{ij}^{(t)}$ be the minimum weight of going from i to j in $\le t$ steps. The Bellman equation says

$$M_{ij}^{(t)} = \min\left(M_{ij}^{(t-1)}, \min_{k:i\to k}\left\{\text{weight}(i \to k) + M_{kj}^{(t-1)}\right\}\right).$$

Let's define a matrix W to store the weights,

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

The nifty thing about this matrix is that it lets us simplify the Bellman equation to

$$M_{ij}^{(t)} = \min_k\left\{W_{ik} + M_{kj}^{(t-1)}\right\}, \qquad M_{ij}^{(1)} = W_{ij}.$$

The first clause in the Bellman equation is taken care of because we defined $W_{ii} = 0$; and the restriction to $\{k : i \to k\}$ is taken care of because we defined $W_{ij} = \infty$ if there is no edge. We could start the iteration at $M^{(0)}$, but it's easy to see that a single iteration of Bellman's equation gives $M^{(1)} = W$, and we have W already, so we might as well use it.

The matrix-Bellman equation can be rewritten as

$$M_{ij}^{(t)} = \left(W_{i1} + M_{1j}^{(t-1)}\right) \wedge \left(W_{i2} + M_{2j}^{(t-1)}\right) \wedge \cdots \wedge \left(W_{in} + M_{nj}^{(t-1)}\right).$$

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 $M^{(t)} = W \otimes M^{(t-1)}$. This nifty notation lets us write out the complete algorithm very concisely:

```
1   Let  M^(1) = W
2   Compute  M^(|V|−1),  using  M^(t) = W ⊗ M^(t−1)
3   Return  M^(|V|−1)
```

As noted above, it's sufficient to compute up to time horizon $|V| - 1$, since we assumed the graph has no negative-weight cycles.

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:

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

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

$$* \; * \; *$$

For interesting problems like Go, or even the Seaquest game shown at the beginning of this section, it's impractical to solve the Bellman equation exactly because the number of states is combinatorially huge. Instead of solving the value function exactly, DeepMind trains a neural network to learn an approximation to the value function.

5.8. Johnson's algorithm

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

- 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 $1/n$.) It measures how 'important' each 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.
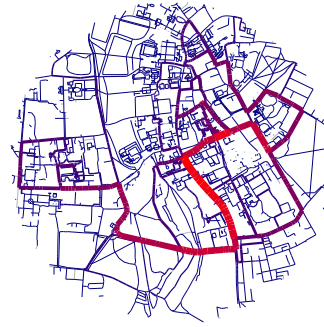
- Each router in the internet has to know, for every packet it might receive, where that packet should be forwarded to. Routers send messages between themselves using the Border Gateway Protocol (BGP), advertising which destinations they know about, and they update their routing tables based on the messages they receive. The entire internet can be thought of as a distributed algorithm for computing all-to-all paths.

We've learnt three algorithms we can use for this purpose: (1) if all edge weights are ≥ 0 we can run Dijkstra's algorithm once from each vertex; (2) we can run Bellman-Ford once from each vertex; (3) we can use dynamic programming with matrices. The running times are

	running time	$E = V - 1$	$E = V(V-1)$	$E = \Theta(V^\alpha)$
Dijkstra	$V \times O(E + V \log V)$	$O(V^2 \log V)$	$O(V^3)$	$O(V^{1+\alpha} + V^2 \log V)$
Bellman-Ford	$V \times O(VE)$	$O(V^3)$	$O(V^4)$	$O(V^{2+\alpha})$
d.p.	$O(V^3 \log V)$	$O(V^3 \log V)$	$O(V^3 \log V)$	$O(V^3 \log V)$
Johnson	same as Dijkstra			

The table shows the running time as a function of V and E, and it also shows it for two special cases, $E = V - 1$ (a tree, the sparsest connected graph on V vertices) and $E = V(V-1)$ (a fully connected graph, the densest graph on V vertices), as well as for $E = \Theta(V^\alpha)$ for $\alpha \in [1, 2]$, which spans the range from sparse to dense. This last column makes it easier to see the comparison. Dijsktra is best for any α, and dynamic programming is better than Bellman-Ford for any $\alpha > 1$.

   The last row is for Johnson's algorithm, the topic of this section. It is as fast as Dijkstra's algorithm, but it also works with positive and negative edge weights. It was discovered by Donald Johnson in 1977.

GENERAL IDEA

Johnson's idea was that we can construct a suitable helper graph, run Dijkstra once from each vertex in the helper graph, and then translate the answers back to the original graph. His method is subtle and clever, but his general strategy is very common, and we'll see it again and again. It's worth highlighting the two parts to his strategy:

TRANSLATION strategy: Translate the problem we want to solve into a different setting, use a standard algorithm in the different setting, then translate the answer back to the original setting. In this case, the translated setting is 'graphs with different edge weights'. Of course we'll need to argue why these translated answers solve the original problem. We'll see more of the TRANSLATION strategy in Section 6.
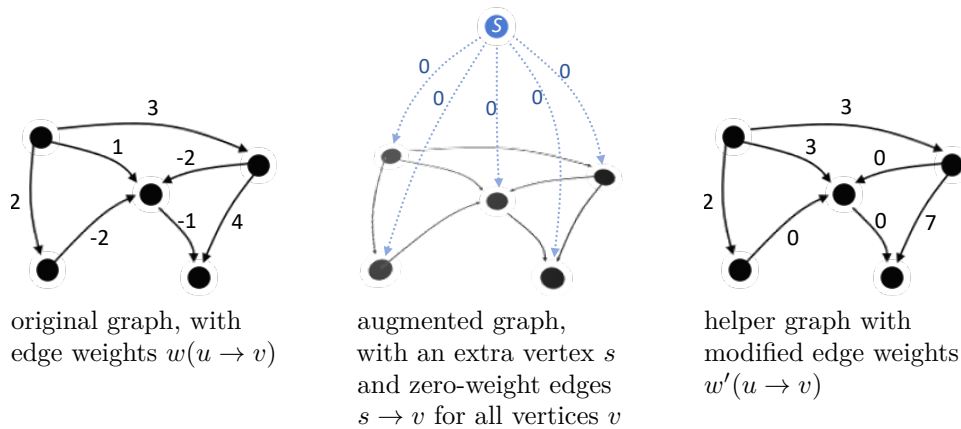
AMORTIZATION strategy: It takes work to construct the helper graph, but this work pays off because we only need to do it once and then we save time on each of the V times that we can run Dijkstra's algorithm rather than Bellman-Ford. We'll see more of the AMORTIZATION strategy in Section 7.

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 augmented graph.   First build an augmented graph with an extra vertex s, as shown below. Run Bellman-Ford on this augmented 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 $w(u \to v)$

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

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

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

3. Dijkstra on the helper graph.   Run Dijkstra's algorithm V times on the helper graph, once from each vertex. (We've ensured that the helper graph has edge weights ≥ 0, so Dijkstra terminates correctly.) CLAIM: Minimum-weight paths in the helper 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 helper graph and in the original graph?

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

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

4. Wrap up.   We've just shown that

$$\begin{array}{ccc} \text{min weight} & & \text{min weight} \\ \text{from } p \text{ to } q & = & \text{from } p \text{ to } q \quad - d_p + d_q \\ \text{in original graph} & & \text{in helper graph} \end{array}$$

which gives us the solution to the problem statement.
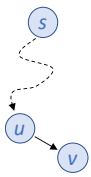
6. Graphs and subgraphs

In this section we will look at algorithms for finding structures within graphs. Here's an example to illustrate.

Example (Minimum spanning trees). Suppose we have to build a power grid to connect 6 cities, and the costs of running cabling are as shown on the left. We've learnt how to find a minimum-cost path between a pair of nodes. But what is the minimum cost *tree* that connects all the nodes? (This is called a minimum spanning tree.)

edge costs   minimum cost path from c to d minimum weight tree
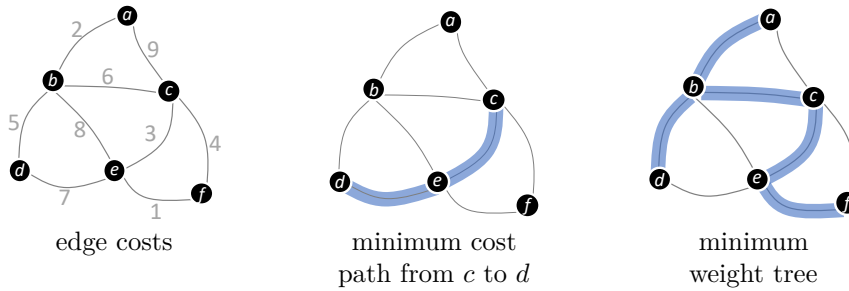
But we're not sudying subgraph algorithms for their own sake. We're studying them to get more because they highlight two strategies for algorithm design:

- We'll see the translation strategy again, in Section 6.4. This is where we translate a problem into a different setting, solve the translated problem using a standard algorithm, and translate the solution back to answer the original problem.

- Often, the soul of an algorithm can be dressed in different guises. For example, the soul of Dijkstra's algorithm can help us find a minimum spanning tree. The algorithms in Sections 6.5–6.7 are different guises for algorithms whose souls we've seen before.

The translation strategy was used by Johnson's algorithm, section 5.8

Prim's algorithm for finding a minimum spanning tree, section 6.5

6.1. Flow networks

For some applications, it's useful to consider a graph where each edge has a capacity. In the internet each link has a maximum bandwidth it can carry; in the road network each road has a maximum amount of traffic it can handle; in an oil pipeline system each pipe has a capacity. Many interesting problems can be boiled down to the question: what is the maximum amount of stuff that can be carried between a given pair of vertices?
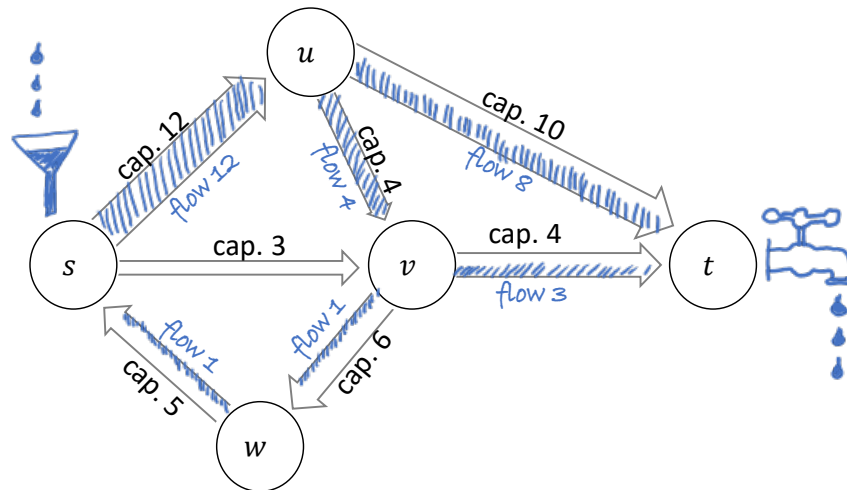
This picture illustrates a flow on a network. There are two distinguished vertices, the *source vertex s* where flow originates, and the *sink vertex t* where flow is consumed. The edges are directed, and labelled with their capacities. The *flow value* is the net flow out of the source vertex, and it's $12 - 1 = 11$ in this picture. This is equal to the net flow into the sink vertex, of course.

What's the maximum possible flow value, over all possible flows? For this simple network, it's fairly easy to discover a flow of value 14. Furthermore, the total capacity of the edges going into the sink is 14, so it's impossible to have a flow of value > 14. Therefore the maximum possible flow value is 14.

In Sections 6.2 and 6.3 we will see an algorithm for finding a maximum flow, and prove that it is correct. First, here is a pair of flow problems[10] that inspired the algorithm.

TWO TRANSPORTATION PROBLEMS

The Russian applied mathematician A.N. Tolstoĭ 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."

We'll only study single-commodity flows, i.e. where there is a single type of 'stuff' flowing. Multi-commodity flow problems are much much harder.

> Exercise. In the standard formulation of the flow problem, there is a single source with unlimited supply capacity, and a single sink. Suppose we have an algorithm that solves this standard problem. Explain how to use it to solve Tolstoĭ's problem.

---

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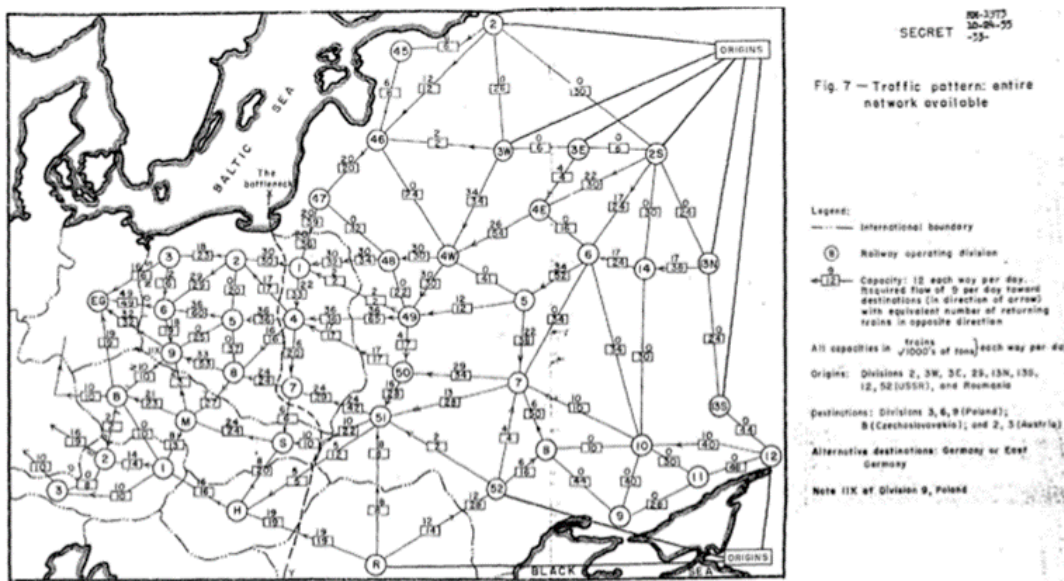
From *Methods of finding the minimum total kilometrage in cargo-transportation planning in space*, A.N.Tolstoy, 1930.

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

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 (vertex EG), they'd need to transport fuel to the front line. The diagram shows the links in the rail network, and the carrying capacity of each link. It also shows the various available fuel sources, aggregated into a single vertex marked ORIGINS. What is the max flow from ORIGINS to EG? More importantly, if the US Air Force wants to strike and degrade one of the links, which link should it target in order to reduce the max flow? It's no use hitting a link where the Soviets can just reroute around the damage.

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).

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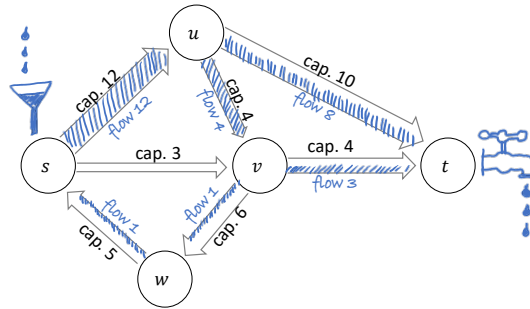
Given a weighted directed graph g with a source s and a sink t, find a flow from s to t with maximum possible value (also called a *maximum flow*).

But to write a proper problem statement, we need to be more precise than this! Here are some definitions. Let the weight associated edge $u \to v$ be $c(u \to v)$, and call this the *capacity* of the edge. Assume it is > 0 for every edge in the graph. A flow is a set of edge labels $f(u \to v)$ such that

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

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 into v as goes out. Flow conservation doesn't need to hold at s or t—indeed, the *value* of a flow is the net flow out of s,
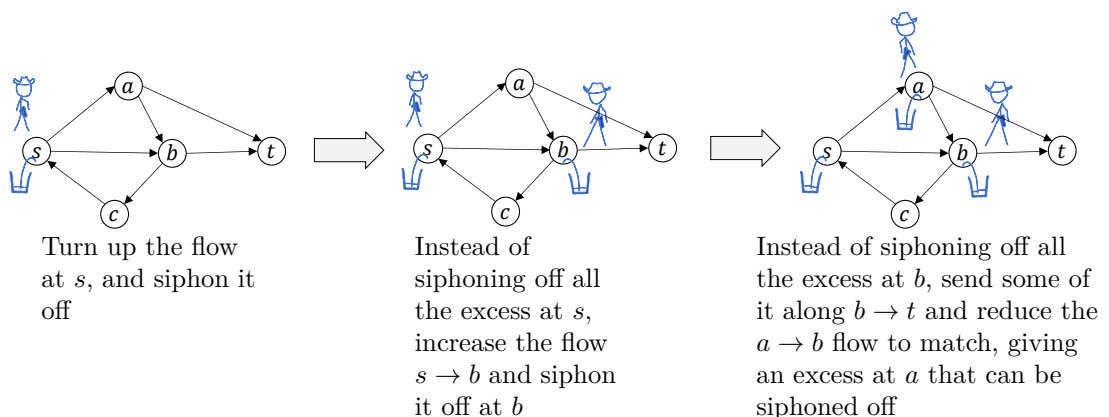
$$\text{value}(f) = \sum_{u:\, s \to u} f(s \to u) - \sum_{u:\, u \to s} f(u \to s).$$

In the network pictured above the flow value is $12 - 1 = 11$, i.e. the total flow out minus the total flow in.

The basic idea of the algorithm is "look for vertices to which we could increase flow".

Imagine that the source and all the other vertices apart from the sink are in bandit country, and the bandits want to siphon off flow from intermediate vertices. We'll assume they can sneak into the vertices to siphon off flow, and to redirect existing flow, and they can also increase the flow at the source. But they daren't do anything that would disrupt the total flow to the sink, because that'd attract the attention of the authorities, who would come and put an end to their banditry. Here are two types of step that the bandits could take, starting from the flow pictured at the top of the page.

Turn up the flow at s, and siphon it off

Instead of siphoning off all the excess at s, increase the flow $s \to b$ and siphon it off at b

Instead of siphoning off all the excess at b, send some of it along $b \to t$ and reduce the $a \to b$ flow to match, giving an excess at a that can be siphoned off

Suppose the bandits are overzealous and they discover that they can siphon off some flow at t. That means that the network operator—which was spying on the bandits all along—has learned a reconfiguration that delivers extra flow to the sink.

Let's look more closely at this reconfiguration. In the network fragment below, the bandits discovered they could siphon off flow at a, thence b, thence c, thence t. How much could they siphon off at each of these locations?
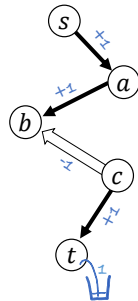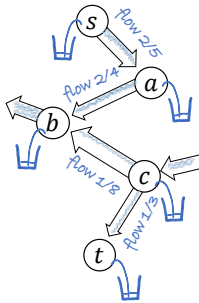
- They could siphon off 3 at a by increasing $s \to a$
- Or siphon off 2 at b by increasing $s \to a \to b$ (limiting factor: spare capacity on $a \to b$)
- Or siphon off 1 at c by increasing $s \to a \to b$ and decreasing $c \to b$, leaving the other outflow at b undisturbed (limiting factor: existing flow on $c \to b$)
- Or siphon off 1 at t by increasing $s \to a \to b$ and decreasing $c \to b$ and increasing $c \to t$, leaving the inflow at c undisturbed (limiting factor: existing flow on $c \to b$)

The network operator only wants to get flow to t, not to any of the other vertices. So it chooses a flow adjustment that gets as much as possible to t, with no excess at any of the other vertices along the path.

The Ford-Fulkerson algorithm starts with an empty flow, then repeatedly uses this 'bandit search' to find whether it's possible to siphon off flow at sink. If it is possible, it adjusts flow along a suitable sequence of edges and thereby increases the flow value. If it's not possible, then the algorithm terminates.

IMPLEMENTATION

There are two pieces that we need to turn into a proper formal algorithm, (1) how exactly the bandit search works, and (2) how to use the results of the bandit search to adjust the flow.

The residual graph.   To formalize the bandit search, we'll build what's called the *residual graph*. This has the same vertices as the flow network, and it has either one or two edges for every edge in the original flow network:

- If $f(u \to v) < c(u \to v)$ in the flow network, let the residual graph have an edge $u \to v$ with the label "increase flow $u \to v$".
- If $f(u \to v) > 0$ in the flow network, let the residual graph have an edge $v \to u$ (i.e. in the opposite direction) with the label "decrease flow $u \to v$".

The two clauses here correspond to the two types of adjustment that the bandits can make. In this illustration, the 'increase' edges are solid lines and the 'decrease' edges are dotted.
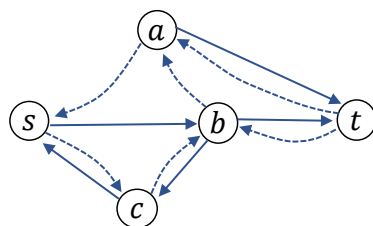

If the bandits can siphon off flow at some vertex u, and if the residual graph has an edge $u \to v$, then they can siphon flow off at v. They can certainly siphon off some flow at s. Thus, if the residual graph has a path from s to t, then they can siphon flow off at t.

Augmenting paths.   Suppose the residual graph has a path from s to t. This is called an
augmenting path. We could find it using breadth-first search or depth-first search, or any
other path-finding algorithm we like.
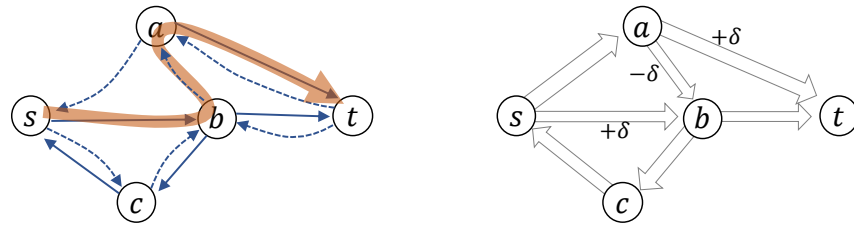
Each edge in the augmenting path has a label, saying either 'increase' or 'decrease'. In the
path shown here, the $s \to b$ edge is labelled "increase the flow $s \to b$", the $b \to a$ edge is
labelled "decrease the flow $a \to b$", and the $a \to t$ edge is labelled "increase the flow $a \to t$".
We pick some amount $\delta > 0$ to increase/decrease each edge by, and we update the flow.
     The crucial thing about this update is that it leaves us with a valid flow. To verify
this, remember the two defining characteristics of a flow: (1) it must satisfy the capacity
constraints $0 \leq f(u \to v) \leq c(u \to v)$, and (2) it must satisfy flow conservation. The
rules for constructing the residual graph ensure that the updated flow satisfies the capacity
constraints, as long as δ is sufficiently small; and by thinking carefully about the four different
possibilities for what happens at each vertex along the augmenting path, we see that the total
flow in minus total flow out is unchanged.
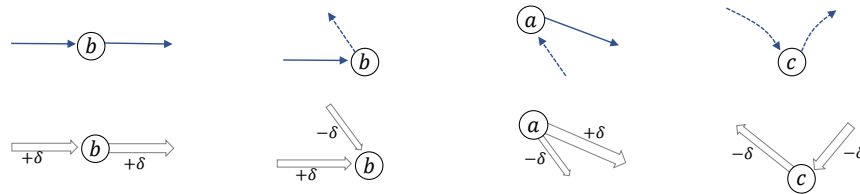
Also, the flow value increases by δ. To see this, consider the two possible labels for the first
edge of the augmenting path. Whether it's an "increase" edge or a "decrease" edge, either
way the net flow out of s increases.


```
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
6       # Define a helper function for finding an augmenting path
7       def find_augmenting_path():
8           # define the residual graph h on the same vertices as g
9           for each edge u → v in g:
10              if f(u → v) < c(u → v): give h an edge u → v labelled "inc"
11              if f(u → v) > 0: give h an edge v → u labelled "dec"
12          if h has a path from s to t:
13              return some such path, together with the labels of its edges
14          else:
15              # There is a set of vertices that we can reach starting from s;
16              # call this "the cut associated with flow f".
17              # We'll use this in the analysis.
18              return None
19
20      # Repeatedly find an augmenting path and add flow to it
21      while True:
22          p = find_augmenting_path()
23          if p is None:
```

```
24                  break  # give up — can't find an augmenting path
25            else:
26                let the vertices of p be s = v_0, v_1, ..., v_k = t
27                δ = ∞ # amount by which we'll augment the flow
28                for each edge v_i → v_{i+1} along p:
29                    if the edge has label "inc":
30                        δ = min(δ, c(v_i → v_{i+1}) − f(v_i → v_{i+1}))
31                    else the edge must have label "dec":
32                        δ = min(δ, f(v_{i+1} → v_i))
33                # assert: δ > 0
34                for each edge v_i → v_{i+1} along p:
35                    if the edge has label "inc":
36                        f(v_i → v_{i+1}) = f(v_i → v_{i+1}) + δ
37                    else the edge must have label "dec":
38                        f(v_{i+1} → v_i) = f(v_{i+1} → v_i) − δ
39                # assert: f is still a valid flow
```

This pseudocode doesn't tell us how to choose the path in line 13. One sensible idea is 'pick the shortest path', and this version is called the Edmonds–Karp algorithm; it is a simple matter of running breadth first search on the residual graph. 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 21: 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 27–32, 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 33, i.e. that $\delta > 0$, by thinking about the residual graph in find_augmenting_path. Therefore the total flow has increased by an integer after lines 34–38. 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 residual graph and find a path in it using breadth first search, so find_augmenting_path is $O(V + E)$. Lines 27–38 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 $O((E+V)f^*)$. 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 worth noting that the running time we found depends on the *values* in the input data (via f^*). This is in contrast to all the algorithms we've seen studied so far, like Quicksort and Depth-first search, in which we found a running time that depends only on the *size* of the data. Often in machine learning and optimization, we get answers that depend on the contents of the data.

On one hand it's good to get an answer that depends on the values in the input data rather than just the size, because any analysis that ignores the data contents can't be very informative. On the other had it's bad in problem because we don't have a useful upper bound for f^*.

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

CORRECTNESS

There are two parts to proving correctness: (1) does this algorithm produce a flow? and (2) is the flow it produces a maximum flow?

We've already argued why the assertion on line 39 is correct, i.e. why the algorithm produces a valid flow at every iteration. The proof that it does indeed produce a maximum flow is left to the next section.

* ⁎ *

In computer science textbooks and on YouTube, there are plenty of explanations of the Ford-Fulkerson algorithm that start by defining the residual graph, and make no mention of what these notes have called the 'bandit search' problem. If you're an engineer and all you want is a recipe to follow, then you don't need to think about the bandit search at all. But if you're a computer scientist or mathematician and you want to understand *why* the algorithm works, the bandit search idea is the linchpin, and the residual graph is just an implementation detail. The big idea in the proof of correctness relates directly to the bandit search problem. The most elegant algorithms are, in my opinion, those in which each part of the algorithm corresponds to a line of a proof, and where the proof is as concise as it can be.

6.3. Max-flow min-cut theorem

In this section we'll prove that the Ford-Fulkerson algorithm finds a maximum flow.
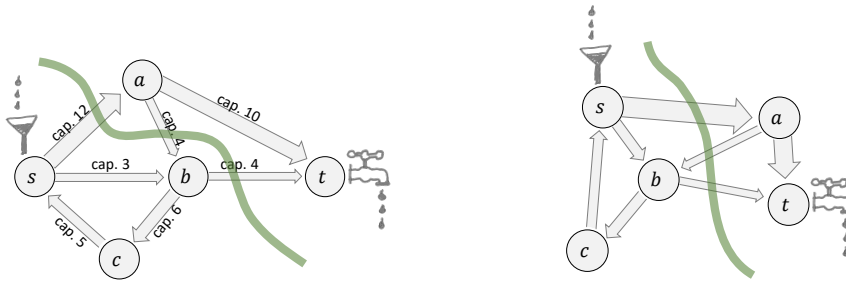
The proof is based on the idea of a *cut*. A cut is a partition of the vertices into two sets, $V = S \cup \bar{S}$, with $s \in S$ and $t \in \bar{S}$. The *capacity* of a cut is

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

The two pictures above shows the same network and cut, with cut capacity 16 (not 20!). The right hand picture emphasizes that a cut splits the vertices into two groups, one group with the source, the other with the sink.

It's obvious that the maximum flow in this network is ≤ 15, since the total capacity of all edges out of the source is $12 + 3 = 15$. Similarly, by considering all the edges into the sink, the maximum flow must be ≤ 14. The idea of a cut is to generalize this type of bound. Looking at the right hand picture above, which shows the cut $(\{s, b, c\}, \{t, a\})$, we see it's impossible to push more flow from left to right than the total left→right capacity, which is $12 + 4 = 16$. If we had chosen the cut $(\{s\}, \{a, b, c, t\})$ we'd get the 'source' bound, that flow ≤ 15, and if we had chosen the cut $(\{s, a, b\}, \{t\})$ we'd get the 'sink' bound, that flow ≤ 14. Here's a theorem to formalize this idea.

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

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

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 $f(u \to v) = c(u \to v) = 0$.

$$\text{value}(f) = \sum_u f(s \to u) - \sum_u f(u \to s) \qquad \text{by definition of flow value}$$

$$= \sum_{v \in S} \left(\sum_u f(v \to u) - \sum_u f(u \to v) \right) \qquad \text{by flow conservation}$$

$$\text{(the term in brackets is zero for } v \neq s)$$

$$= \sum_{v \in S} \sum_{u \in S} f(v \to u) + \sum_{v \in S} \sum_{u \notin S} f(v \to u)$$

$$- \sum_{v \in S} \sum_{u \in S} f(u \to v) - \sum_{v \in S} \sum_{u \notin S} f(u \to v)$$

$$\text{(splitting the sum over } u \text{ into two sums, } u \in S \text{ and } u \notin S \text{)}$$

$$= \sum_{v \in S} \sum_{u \notin S} f(v \to u) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) \qquad \text{by 'telescoping' the sum}$$

$$\leq \sum_{v \in S} \sum_{u \notin S} f(v \to u) \qquad \text{since } f \geq 0 \tag{1}$$

$$\leq \sum_{v \in S} \sum_{u \notin S} c(v \to u) \qquad \text{since } f \leq c \tag{2}$$

$$= \text{capacity}(S, \bar{S}) \qquad \text{by definition of cut capacity.}$$

This completes the proof.   □
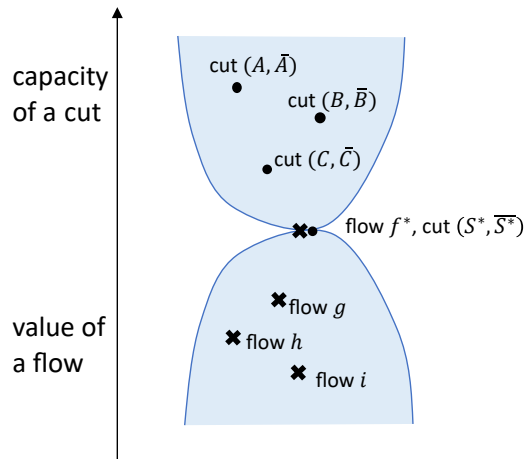
What's this theorem for? The theorem says that for *any* cut, *every* possible flow's value is $\leq$ that cut's capacity. Likewise, for any flow, every possible cut's capacity is $\geq$ that flow's value.

Hence, if we're able to find a flow f^* and a matching cut $(S^*, \bar{S}^*)$ such that value$(f^*) =$ capacity$(S^*, \bar{S}^*)$, then every other flow must have value $\leq$ capacity$(S^*, \bar{S}^*)$, therefore f^* is a maximum flow. In other words, if we can find a matching flow and cut, then the cut acts as a 'certificate of correctness'. If someone doubts whether our proposed flow truly is a maximum flow, all we need do is show them our cut, and (assuming the flow value is equal to the cut capacity!) that proves our flow is correct.[11]

This is exactly what we need to prove the Ford-Fulkerson algorithm correct. The algorithm runs the 'bandit search' to find all the vertices to which flow could be increased. The algorithm terminates when it can't increase flow to the sink—in other words, when the bandit search produces a cut. And this cut is exactly what we need to certify that the flow it just found is a maximum flow!

Theorem (Correctness of Ford-Fulkerson). *Suppose the algorithm terminates, and f^* is the final flow it produces. Then f^* is a maximum flow.*

Proof. Let S^* be the set of vertices found in the final call to find_augmenting_path on line 16, page 32. Since it's the final call, we know it failed to find a path to the sink, i.e. $t \notin S^*$, hence $(S^*, \bar{S}^*)$ is a cut.

Now imagine drawing the original flow network with all the S^* vertices on the left and all the $\bar{S}^*$ vertices on the right. Suppose the network has an edge $u \to v$ that goes from left to right, i.e. $u \in S^*$ and $v \notin S^*$. This means that the residual graph has a path from s to u, but not a path from s to v; therefore $u \to v$ cannot be present in the residual graph. Hence, by the condition on line 10, $f(u \to v) = c(u \to v)$.

Next, suppose the flow network has an edge $u \to v$ that goes from right to left, i.e. $v \in S^*$ and $u \notin S^*$. This means that the residual graph has a path from s to v, but not a path from s to u, hence $v \to u$ is not present in the residual graph. Therefore, by line 11, $f(u \to v) = 0$.

We have proved that for any edge from S^* to $\bar{S}^*$, the flow on that edge is equal to the capacity, hence inequality (2) in the proof of the max-flow min-cut theorem is an equality. And we have also proved that for any edge from $\bar{S}^*$ to S^*, the flow on that edge is equal to 0, hence inequality (1) is an equality also. Therefore the derivation in that proof shows that value(f^*) is equal to capacity$(S^*, \bar{S}^*)$.

As we have argued, the cut $(S^*, \bar{S}^*)$ thus acts as a 'certificate' proving that flow f^* is a maximum flow.   □

<p style="text-align:center">∗ ∗ ∗</p>

---

[11]In the practical assignment for this part of the course, you're asked to produce a maximum flow and a matching cut. The tester doesn't bother computing its own answer to verify against yours, it simply checks the certificate you provided.

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

6.4. Matchings

There are several graph problems that don't on the surface look like flow networks, but which can be solved by translating them into a well-chosen maximum flow problem. Finding matchings in bipartite graphs is one example. The example sheet has more.
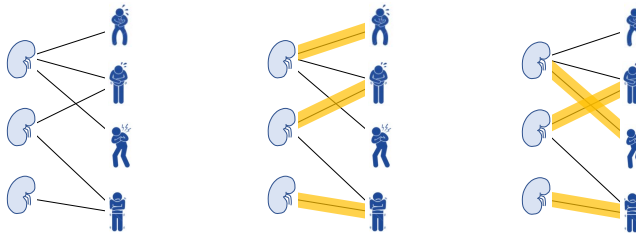

A *bipartite graph* is one in which the vertices are split into two sets, and all the edges have one end in one set and the other end in the other set. For example, kidney transplant donors and recipients, with edges to indicate compatibility. We'll assume the graph is undirected.

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. The *size* of a matching is the number of edges it contains. A *maximum matching* is one with the largest possible size. Our goal is to find a maximum matching.

IMPLEMENTATION

Let's translate the matching problem into a flow problem, as follows.

1. start with a bipartite graph
2. create a helper graph as follows: 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 on the helper graph to find a maximum flow from s to t
4. interpret that flow as a matching in the original bipartite graph
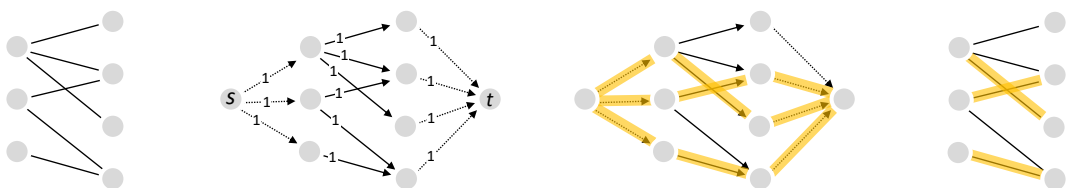
ANALYSIS

It's easy to not even notice that there's something that needs to be proved here. To understand the issue, it's helpful to imagine we're explaining the procedure to someone who doesn't know anything at all about Ford–Fulkerson. They'll ask us two questions:

- How do you know that a maximum flow can be translated into a matching? For example, what if it returns a flow with a fractional amount on some edge?
- How do you know that your maximum flow actually gives a maximum matching? What if there's a larger-size matching out there, which perhaps corresponds to a *lower*-value flow, or perhaps doesn't correspond to a flow at all?
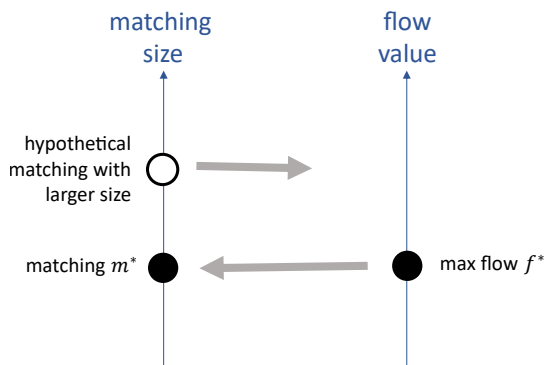
We need to justify the translation in two directions. First we have to justify why the flow f^* found in Step 3 can be translated into a matching m^*. Second we have to justify why any hypothetical larger-size matching m' would translate into a higher-value flow f'.

Once we've justified these translations, it's easy to argue that m^* is a maximum-size matching. Suppose it were not. Then there would be a larger-size matching m', with a corresponding f' for which value$(f') >$ value(f^*). But f^* is a maximum flow, therefore no such f' can exist. Hence the premise is false, i.e. m^* is a maximum-size matching.

This style of algorithm and proof is called the TRANSLATION strategy. Remember, when you use it, that you have to justify the translation in both directions. Once we've figured out what it is we have to prove, the proof is easy.

Lemma (translation matching $\leftrightarrow$ flow).
1. *Step 3 of the algorithm described above terminates, and the flow f^* that it produces can be translated into a matching m^*, with* size$(m^*) =$ value(f^*).
2. *Any matching m translates into a flow f with* size$(m) =$ value(f). *In particular, if there were any matching m' with* size$(m') >$ size(m^*) *then its corresponding flow f' would have* value$(m') >$ value(m^*).

Proof (of 1). The lemma in Section 6.2 on page 33 tells us that the Ford-Fulkerson algorithm terminates, since all edge capacities are integer. 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 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 2). Take any matching m and translate it into a flow f in the natural way, i.e. with a flow of 1 from s to every matched left hand vertex, and similarly for t. It's easy to use the definition of 'matching' to prove that f is indeed a flow, i.e. that it satisfies the capacity constraints as well as flow conservation. From this translation it's clear that size$(m) =$ value(f).   □

6.5. Prim's algorithm

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

Given a connected undirected graph g with edge weights, a *spanning tree* of g is a tree that connects all of g's vertices, using some or all of g's edges. A *minimum spanning tree* (MST) is a spanning tree that has minimum weight among all spanning trees. (The *weight* of a tree is just the sum of the weights of its edges.)
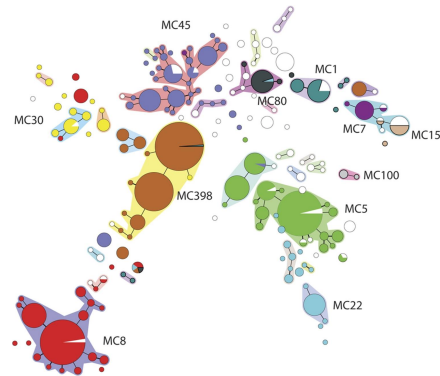
How can we find an MST? We'll look at an algorithm due to Jarnik (1930), and independently to Prim (1957) and Dijkstra (1959).

undirected graph with edge weights

a tree, but not spanning

spanning tree of weight 6

spanning tree of weight 6
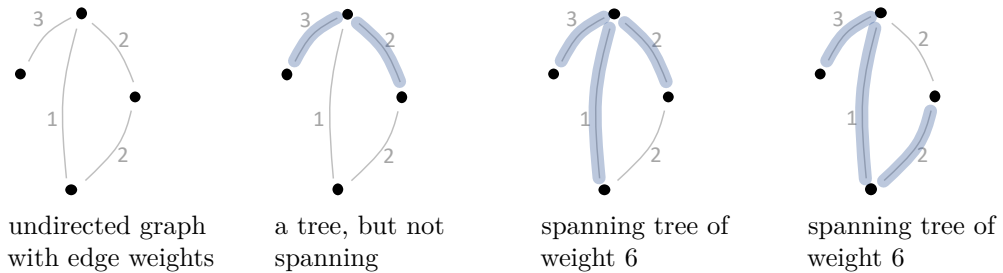
APPLICATIONS

- 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?

The 'edit distance' between two strings is a measure of how different they are. See Section 1.2.

- Minimal spanning trees are a useful tool for exploratory data analysis. In this illustration from bioinformatics[12], each vertex is a genotype of *Staphylococcus aureus*, and the size shows the prevalance of that genotype in the study sample. Let there be edges between all genotypes, weighted according to edit distance. The illustration shows the MST, after some additional high-weight edges are removed.


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 adjacent vertices that aren't part of the tree, pick the edge of lowest weight among these and add it to the tree, then repeat.

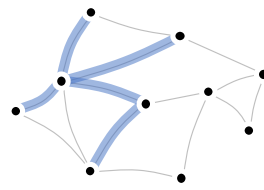This greedy algorithm will certainly give us a spanning tree. To prove that it's a MST takes some more thought.

---

[12]From *Multiple-Locus Variable Number Tandem Repeat Analysis of Staphylococcus Aureus*, Schouls et al., PLoS ONE 2009.
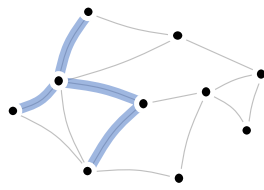
a tree build up with four edges so far
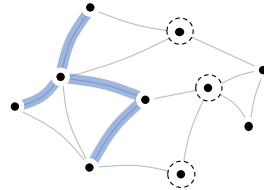
three candidate vertices to add next

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

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. For each of the frontier vertices w, we'll store the lowest-weight edge connecting it to the tree that we've discovered so far (w.come_from), and the weight of that edge (w.distance). We pick the frontier vertex v with the smallest v.distance, add it to the tree, and add its neighbours to the frontier if they're not already in the tree. When the algorithm terminates, an MST is formed from the edges

$$\{v \leftrightarrow v.\text{come\_from} \ : \ v \in V \setminus \{s\}\}.$$

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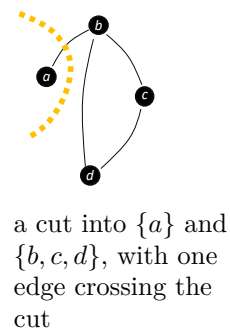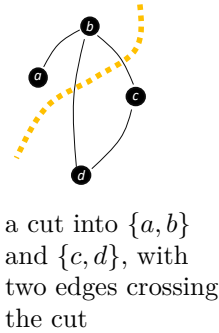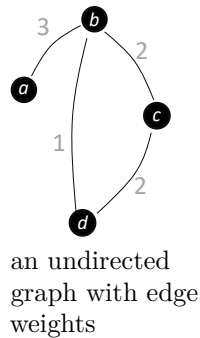
Compared to Dijkstra's algorithm, we need some extra lines to keep track of the tree (lines labelled +), and two modified lines (labelled ×) because here we're interested in 'distance from the tree' whereas Dijkstra is interested in 'distance from the start node'. The start vertex s can be chosen arbitrarily.

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 |

It can be proved by induction that Prim's algorithm produces an MST, using the following theorem. The details of the induction are left as an exercise.

<div style="float:left">The theorem as stated here is more general than is needed for Prim's algorithm. That's so that we can re-use it for Kruskal's algorithm in the next section.</div>

Theorem.   *Suppose we have a forest F and a cut C such that (i) no edges of F cross C, and (ii) there exists some MST that contains F. Then, if we add to F a min-weight edge that crosses C, the result is still part of a MST.*

The following proof is not examinable. It is all maths, no algorithm.

Proof. Let F be the forest, 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 forest F and the cut C | a MST $\bar{F}$ | a different MST $\hat{F}$ |
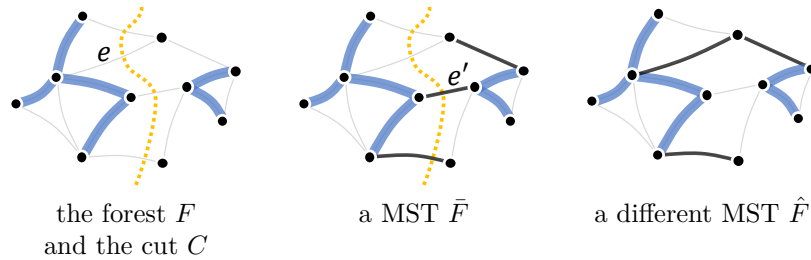
Suppose then that $\bar{F}$ doesn't contain e. Let u and v be the vertices at either end of e, and consider the path in $\bar{F}$ between u and v. (There must be such a path, since $\bar{F}$ is a spanning tree, i.e. it connects all the vertices.) This path must cross the cut (since its ends are on different 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 connected. If this claim is true, then $\hat{F}$ must be a tree, since it has the same number of edges as $\bar{F}$ namely $|V| - 1$. Therefore it is a MST including $F \cup \{e\}$, and the theorem is proved.

    PROOF OF CLAIM. Pick any two vertices x and y. Since $\bar{F}$ is a spanning tree, there is a path in $\bar{F}$ from x to y. If this path doesn't use e' then it is a path in $\hat{F}$, and we are done. If it does use e', consider the paths $x - y$ and $u - v$ in $\bar{F}$, both of which use e'. Either x and u are on one side of e' and y and v are on the other (as illustrated), or vice versa. Either way, we can build a path $x - y$ using e instead of e', by splicing in the relevant part of the $u - v$ path. This new path we've built is in $\hat{F}$. Since $\hat{F}$ contains a path between any two vertices x and y, it is connected. □
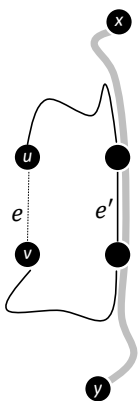
6.6. Kruskal's algorithm

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 that are rather useful in data science.

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.
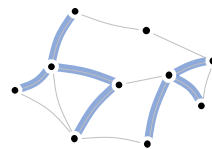
Kruskal's algorithm maintains a 'forest'. Look back at Section 5.1 for the definition.
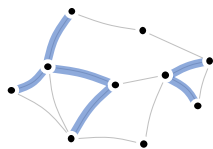
four tree fragments have been found so far, including two trees that each consist of a single vertex
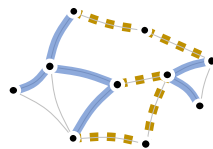
five candidate edges that would join two fragments

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

PROBLEM STATEMENT

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

IMPLEMENTATION

We could scan through all the edges in the graph at every iteration, looking for the best edge to add next. Or we could maintain a list of all candidate edges, pre-sorted in order of increasing weight, and iterate through it; and every time we join two fragments, remove all the edges from this list that have just become redundant.

Kruskal's algorithm doesn't do either of these. It uses a list of edges pre-sorted in order of increasing weight, but it doesn't do any housekeeping on the list—it just considers each edge in turn, asks "does this edge join two fragments?", and skips over the edges that don't. The data structure it uses to perform this test is called a DisjointSet. This keeps track of a collection of disjoint sets (sets with no common elements), also known as a partition. Here we're using it to keep track of which vertices belong to which fragment.

We'll study the DisjointSet data structure in Section 7.9

Initially (lines 4–5) every vertex is in its own fragment. We iterate through all the edges of the graph in order of edge weight, lowest edge weight first (lines 6–8), and for each edge we test whether that edge's ends are belong to the same set (lines 9–11). If they belong to different sets, we add that edge to the tree and merge the two sets (lines 12–13).

```
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
17        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[13]. 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 something that's part of an MST.
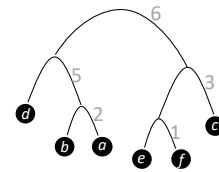
APPLICATION

If we draw the tree fragments another way, the operation of Kruskal's algorithm looks like clustering, and its intermediate stages correspond to a classification tree:
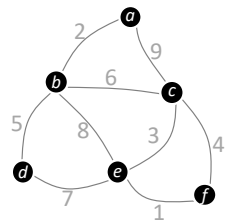
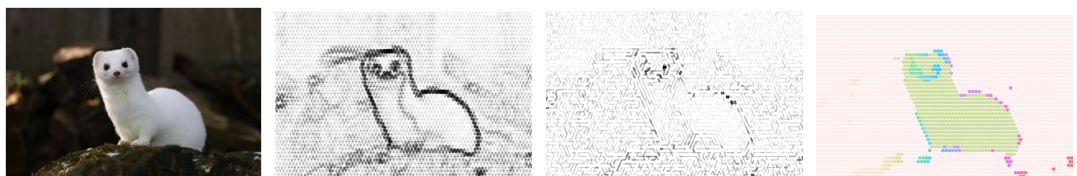an undirected graph
with edge weights
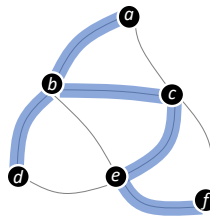
the MST found by
Kruskal's algorithm

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.

---

[13]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.

6.7. Topological sort

A directed graph can be used to represent precedence or preference.
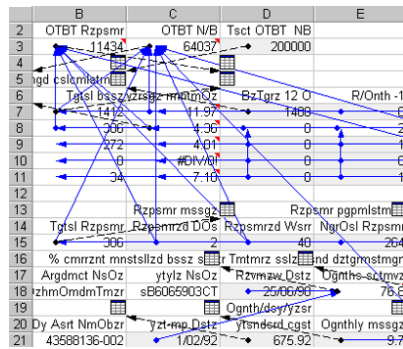

The prototypical application, for computer scientists, is representing computational dependencies. In an Excel spreadsheet, for example, imagine a graph with a vertex for each cell and edges between cells to indicate "cell v is a function of cell w, so w needs to be evaluated before v" (you can see these edges with the menu option 'Show Dependencies'). Excel needs to figure out an order in which to evaluate all the cells, and this order needs to respect the edge directions. Think of it as putting all the cells in a straight line, such that the edges of the dependency graph all point in the same direction. Such an ordering is called a *total order* or *linear order*.
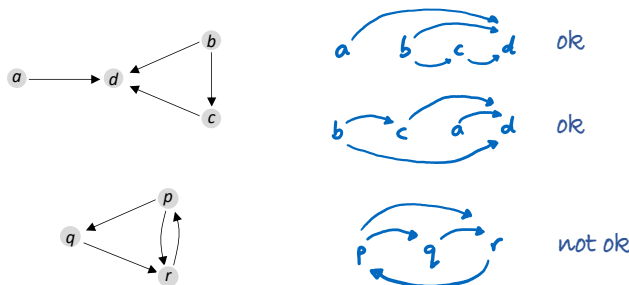
For what graphs it possible to put all the vertices into a total order? And, if it is possible, how can we compute the total order? The picture above shows two simple graphs, and three attempted total orders. The first two are valid total orders, and the third is not—and a moment's thought about the second graph tells us that it's impossible to find a total order, because of the cycle.

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 5.2, but modified so that it visits the entire graph (rather than just the part reachable from some given start vertex).

```
1   def dfs_recurse_all(g):
2       for v in g.vertices:
3           v.visited = False
4       for v in g.vertices:
5           if not v.visited:
6               visit(v)  # start dfs from v
7
8   def visit(v):
9       v.visited = True
10      for w in v.neighbours:
11          if not w.visited:
12              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 f. Here is the flame chart for running dfs_recurse_all on a simple graph.
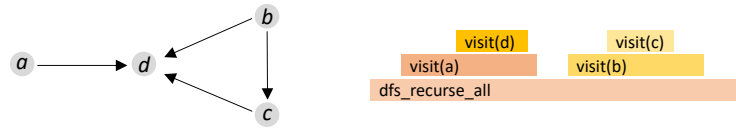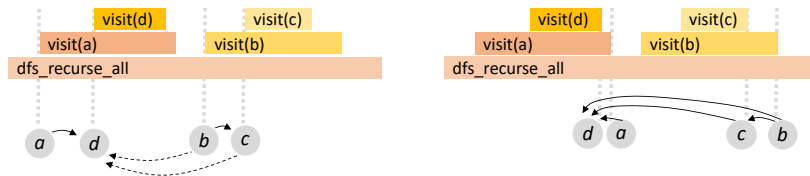

And here, on the left, is what happens if we order vertices by when we visit them. It turns out not to be a total order. A better guess is to order vertices by when visit(v) *returns*.

PROBLEM STATEMENT

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

Why the restriction to DAGs? Recall the definition. 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. If a graph is not a DAG then it has a cycle, and as we noted at the beginning of this section that means no total ordering is possible. It turns out that every DAG has a total ordering, as we'll see from the proof of correctness below.

Don't get muddled by the word 'acyclic'. A DAG doesn't have to be a tree! The graph we ran dfs_recurse_all on is a DAG.

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.

Running time.   We haven't changed anything substantial from dfs_recurse so the analysis in Section 5.2 still applies: the algorithm obviously terminates (thanks to the visited flag, which ensures we never visit a vertex more than once), and its running time is $O(V + E)$.

Theorem (Correctness).   *The* toposort *algorithm 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. □

$*\ *\ *$

The breakpoint proof technique.   The proof technique we used here was (1) consider an instant in time at which the algorithm has just reached a line of code; (2) reason about the current state of all the variables, and the call stack, using mathematical logic; (3) make an inference about what the algorithm does next. This is the same structure as the proof of correctness of Dijkstra's algorithm.

For this proof technique to work, we may need to store extra information about program state, so that the mathematical reasoning can use it. In this case, we invented the variable v.colour, which records a useful fact about what happened in the past. The algorithm doesn't need it, but it's useful for the maths proof.

7. Advanced data structures

7.1. Aggregate analysis

When we design an algorithm, it's good practice to find its worst-case running time, and we typically express it using big-O notation. But when we design a data structure it's more useful to know the worst-case running time for a *sequence* of operations. Here are some examples to illustrate why.

Example (Dijkstra's algorithm / priority queue). Dijkstra's algorithm uses the Priority Queue data structure, which supports operations popmin, push, and decreasekey. A single run of Dijkstra's algorithm involves

$$O(V) \times \mathsf{popmin} + O(E) \times \mathsf{push} \ / \ \mathsf{decreasekey}.$$

To work out the worst-case cost of a run of Dijkstra's algorithm, it doesn't matter what the worst-case cost is for a *single* call to popmin or the other operations, what matters is the worst-case cost for the *aggregate* of all of the calls. ◇

But can't we find the worst-case cost for the aggregate of many calls by simply adding up the worst-case costs for each individual call? The next example shows why not. It shows that if we simply add up worst-case costs of individual operations, we'll get an unduly pessimistic bound for the worst-case cost of the aggregate.

Example (Binomial heap). Suppose we have a binomial heap and we know it never has more than N elements. The worst-case cost of inserting an element is $O(\log N)$, for the case when there are $O(\log N)$ trees in the heap.
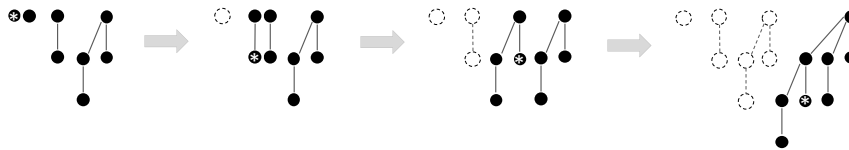
But what about *two* insertions in a row? We certainly can't hit this $O(\log N)$ worst case twice in a row. Indeed, with some careful though about how binomial heaps work, either the first insertion or the second insertion must be $O(1)$. With even more careful thought it can be shown that the aggregate cost of N insertions is $O(N)$.

Another way to put it: if one operation ends up needing to do a lot of work, then it will at least leave the data structure in a nice clean state so that the following operations are fast. ◇

Example (HashTable). A hash table is typically stored as an array, and we increase the capacity of the array and reshash all the existing items whenever the occupancy exceeds a certain threshold. Java's HashMap, for example, rehashes once occupancy reaches 75%. Most insertions don't trigger a rehash so they are very fast, $O(1)$ for a hash table with chaining; but every so often an insertion will trigger a rehash, which takes $O(N)$ where N is the number of items. So the worst case for a single insertion is $O(N)$—but the worst case for N insertions grows slower than N^2. ◇

Advanced data structures are those that have been designed with cunning tricks so that the work done by one operation can benefit subsequent operations. This is done to reduce the aggregate cost of sequences of operations.

The study of advanced data structures is centered around analysing the worst-case aggregate computational complexity of sequences of operations. This is known as *aggregate analysis.*

TIGHT BOUNDS AND WORST CASES

When we analyze computational complexity, it's good practice to give *tight* big-O bounds on the worst-case performance.

It's technically correct to say that the worst-case cost of inserting N elements into a binomial heap, starting from empty, is $O(N \log N)$. It's also $O(N!)$, and $O(N^{N^N})$ ... these are all technically correct, but pretty useless! What's the worst case that can *actually happen*? After looking at some example binomial heaps, we soon find we're unable to construct a scenario in which the cost is $N \log N$; this tells us that either the $O(N \log N)$ bound isn't tight, or that we haven't looked hard enough.

refresher of
section 2.3.2 Formally, big-O notation provides an upper bound. When we say "the worst-case cost is $O(f(N))$" what we really mean is: there exist N_0 and $\kappa > 0$ such that for all $N \geq N_0$,

$$\begin{array}{l} \text{worst-case cost for} \\ \text{problem of size } N \end{array} \leq \kappa f(N).$$

To argue that this bound is tight, we should be able to demonstrate scenarios of cost at least $f(N)$. Formally[14], we should give a constant $\kappa' > 0$, and a sequence of scenarios $i = 1, 2, \dots$ of sizes $N_1, N_2, \dots$ with $N_i \to \infty$ such that

$$\begin{array}{l} \text{cost for} \\ \text{scenario } i \end{array} \geq \kappa' f(N_i).$$

Only if both of these bounds hold would we say that our big-O bound is tight.

---

[14]The purists who want to say that the worst-case complexity is $\Theta(f(N))$ would need to give N_0 and $\kappa' > 0$ and a scenario for *every* $N \geq N_0$ of cost $\geq \kappa' f(N)$. In the author's opinion, that's going beyond the call of duty.

7.2. Amortized costs: introduction

"Amortize a loan" means "pay it off gradually by making regular repayments before the debt finally becomes due". In computer science, amortization is a cunning accounting trick for reasoning about aggregate costs, based on pretending that one operation can "pay off" the running time of a subsequent operation.

Here's a simple illustration. Suppose we want to store a list of items, and we have to support four operations:

```
class MinList<T>:
    append(T v)   # add a new item v to the list
    flush()       # empty the list
    foreach(f)    # do f(x) for each item in the list
    T min()       # get the minimum of all items in the list
```

The first three operations are straightforward; a simple linked list will do. To implement min, here are four stages of enlightenment.

Stage 0. Simply iterate through the entire list every time we want to compute min. This takes time $\Theta(N)$, where N is the number of items in the list.

Stage 1. It's a waste to redo the work involved in computing min. Instead, we should remember the result of the last call to min, and keep a pointer to the tail of the list at the time of that last call. Next time we need min we only need to iterate through the items added after the last min or flush. It's faster—but the worst case is still $\Theta(N)$.

Stage 2. We could store the minimum of the entire list, and update it every time a new item is added. This way, append and min are both $O(1)$, and obviously we can't do any better than this! We'd describe this as "amortizing the computation of min", meaning we split the work up into small pieces done along the way.

Stage 3. If we count up the total amount of work for the Stage 2 implementation, it's no better than that for Stage 1. (It could even be worse, if flush is called before min.) Morally, it's unfair to say that Stage 2 has better running-time complexity. Instead, we should just stick with the Stage 1 implementation and 'pay off' the running time of min in early repayments, by *ascribing* its cost to the append calls that preceded it, and which caused min all that work in the first place.
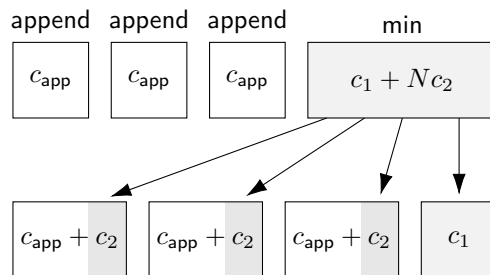
We'd say "the amortized cost of append is $c_{app} + c_2$, and the amortized cost of min is c_1".

The point of this accounting trick is to make it easier to get tight bounds on aggregate costs.

Consider the Stage 1 implementation, and suppose we perform a sequence of $m_1 \times$ append interspersed with $m_2 \times$ min, on an initially-empty list. According to the naive worst-case analysis,

> *The cost of each append is $c_{app} = O(1)$. The worst-case cost of min is $O(N)$ where $N = m_1$ is the maximum number of items in the list. So the worst-case aggregate cost is $O(m_1 + m_2 m_1)$.*

This isn't a tight bound. If we think more cleverly about aggregate costs, and about how many times each of the m_1 items gets 'touched' by min, we deduce that the aggregate cost is actually $O(m_1 + m_2)$. The point of amortized costs is to let us get to the same answer but with less cleverness:

The amortized cost of each append is $c_{app} + c_2 = O(1)$. The amortized cost of each min is $c_1 = O(1)$. So the worst-case aggregate cost is $m_1 O(1) + m_2 O(1) = O(m_1 + m_2)$.

It's a nuisance to always have to reason carefully about aggregate costs of sequences of operations. Much better if someone (preferably someone else!) thinks hard about the best way to ascribe costs, and tells us an amortized cost for each operation that the data structure supports. Then it's easy for us to work out aggregate costs, by just adding up amortized costs. Amortization is nothing more than an accounting trick to make it easier to find tight bounds on aggregate cost.

There's a subtle caveat. In the analysis we just did, we snuck in the little phrase 'on an initially-empty list'. Suppose instead we had started with a list of N items, and we called min once: then the cost of this single operation is N, and no clever accounting trick can turn it into $O(1)$.

In practice, this isn't a problem. For example, when we set out to analyse Dijkstra's algorithm, which uses a Priority Queue, we do indeed start out with an initially-empty data structure. It's only when we start thinking too hard about what amortized analysis actually means that things get confusing—and the best way to resolve confusion is with a rigorous definition. That's the topic of the next section.

7.3. Amortized costs: definition

We use amortized costs to reason about aggregate costs of sequences of operations.

     We might read for example that a certain data structure "supports push at amortized cost $O(1)$, and popmin at amortized cost $O(\log N)$". This is just a way of writing a statement about aggregate costs: it says that for any sequence comprising m_1 push operations and m_2 popmin operations, applied to an initially-empty data structure,

$$\begin{array}{c}\text{worst-case}\\\text{aggregate cost}\end{array} \;\le\; m_1 O(1) + m_2 O(\log N) \;=\; O(m_1 + m_2 \log N).$$

FUNDAMENTAL INEQUALITY OF AMORTIZED ANALYSIS

Here is the formal definition. Let there be a sequence of m operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \ldots, c_m$. Suppose someone invents $c_1', c_2', \ldots, c_m'$ such that

$$c_1 + \cdots + c_j \;\le\; c_1' + \cdots + c_j' \quad \text{for all } j \le m\,;$$

then we call these *amortized costs*. In words,

$$\begin{array}{c}\text{aggregate true cost of a}\\\text{sequence of operations}\end{array} \;\le\; \begin{array}{c}\text{aggregate amortized cost}\\\text{of those operations.}\end{array}$$

 This is the *fundamental inequality of amortized analysis*. If someone else has told us amortized costs, then this inequality tells us what we can do with them. If we are asked to find amortized costs, this is the inequality we have to ensure is satisfied.

     How does one invent amortized costs? Section 7.4 suggests a useful strategy—but for now we'll just take them as given.

Asymptotic usage.   The fundamental inequality applies to sequences of any length. It is *not* an asymptotic (big-O) statement.

     Nevertheless, it is in the context of asymptotic analysis that we usually encounter amortized costs. We might read for example "the amortized cost of push is $O(1)$ and the amortized cost of popmin is $O(\log N)$, where N is an upper bound on the number of elements stored". This tells us that for any sequence of $m_1 \times$ push and $m_2 \times$ popmin operations, applied to an initially empty data structure,

$$\begin{array}{c}\text{worst-case}\\\text{aggregate cost}\end{array} \;\le\; m_1 O(1) + m_2 O(\log N) \;=\; O(m_1 + m_2 \log N).$$

More precisely: there exists N_0 and $\kappa > 0$ such that, for any $N \ge N_0$ and for any such sequence of operations on a data structure which always has $\le N$ elements,

$$\begin{array}{c}\text{worst-case}\\\text{aggregate cost}\end{array} \;\le\; \kappa(m_1 + m_2 \log N).$$

Typically we're interested in the case where m_1 and m_2 grow with N. It might be $m_1 = N^2$ and $m_2 = 1$, or $m_1 = \log N$ and $m_2 = \log \log N$, or anything at all—the fundamental inequality has to hold for all sequences.
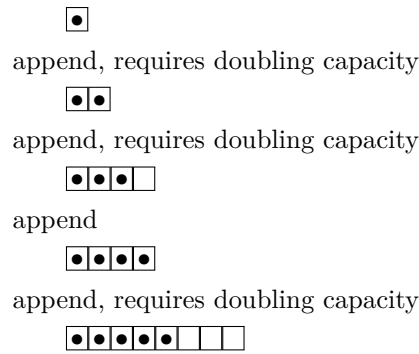
EXAMPLE: DYNAMIC ARRAY

Here is a more involved example. Consider a dynamically-sized array with initial capacity 1, and which doubles its capacity whenever it becomes full. (To be precise: we maintain a fixed-length array; when it becomes full then we allocate a new fixed-length array of double the length, copy everything across from the old array, and deallocate the old array. We'll only consider appending, not deleting.)

<div align="center">

initially empty

□

append

</div>

$$\boxed{\bullet}$$

append, requires doubling capacity

$$\boxed{\bullet\,\bullet}$$

append, requires doubling capacity

$$\boxed{\bullet\,\bullet\,\bullet\,}$$

append

$$\boxed{\bullet\,\bullet\,\bullet\,\bullet}$$

append, requires doubling capacity

$$\boxed{\bullet\,\bullet\,\bullet\,\bullet\,\bullet\,\,\,}$$

Suppose that the cost of writing in an element is 1, and the cost of doubling capacity from m to $2m$ and copying everything across is κm for some constant $\kappa > 0$.

Aggregate analysis.   After adding n elements, the cost of the initial writes is n, and the total cost from all of the doubling is

$$\kappa\big(1 + 2 + \cdots + 2^{\lfloor \log_2(n-1)\rfloor}\big)$$

which is $\leq \kappa(2n - 3)$. Thus, the total cost of n calls to append is $\leq \kappa(2n - 3) + n = O(n)$.

Amortized costs.   Let's ascribe a cost $c' = 2\kappa + 1$ to each append operation. Then, for n calls to append,

$$\text{aggregate true cost} \leq n(2\kappa + 1) - 3\kappa\,,$$
$$\text{aggregate amortized cost} = n(2\kappa + 1)$$

so the fundamental inequality "aggregate true cost $\leq$ aggregate amortized cost" is satisfied, i.e. these are valid amortized costs. We'd write this as "the amortized cost of append is $O(1)$."

$$* \;\; * \;\; *$$

Why do we keep on putting in the caveat "applied to an initially empty data structure"? It's easier to explain why this is needed, now that we have a rigorous definition and a concrete example.

Suppose we start with a dynamic array that's on the brink of needing to be expanded. In other words, let it have capacity N, and let it hold N items. (Of course this requires that N be a power of 2.) Now, consider a single append: this single operation has cost $\kappa N + 1$. If we wanted our fundamental inequality to hold for all possible sequences of operations and all possible initial states of the data structure, we'd be forced to say that the amortized cost of append is $O(N)$—but then we'd be back in the land of naive per-operation worst case analysis, and that doesn't give us tight bounds for other sequences of operations.

This is why formal statements about amortized costs generally have awkward phrasing such as "consider any sequence of m operations on a data structure, initially empty, whose size is always $\leq N$", or alternatively "consider any sequence of $m \geq N$ operations on a data structure, initially empty, where N of those operations are insertions".

7.4. Potential functions

How do we come up with amortized costs? There's a systematic approach that sometimes gives useful answers, based on *potential functions.*

Given a data structure, there are many possible states it might be in. Here, 'state' includes absolutely everything: its capacity, its contents, how those contents are arranged, which bits have pointers to which other bits, and so on. Let Ω be the set of all possible states. A function $\Phi : \Omega \to \mathbb{R}$ is called a *potential function* if

$$\Phi(\mathcal{S}) \geq 0 \quad \text{for all } \mathcal{S} \in \Omega, \qquad \Phi(\mathsf{empty}) = 0.$$

Here empty refers to the empty intial state. Now, consider an operation *op* which, when applied to state $\mathcal{S}_{\mathrm{ante}}$, yields state $\mathcal{S}_{\mathrm{post}}$, and which has true cost c. We'll write this as

$$\mathcal{S}_{\mathrm{ante}} \xrightarrow{\ c\ } \mathcal{S}_{\mathrm{post}}.$$

Define the modified cost c' of *op* to be

$$c' = c + \Phi(\mathcal{S}_{\mathrm{post}}) - \Phi(\mathcal{S}_{\mathrm{ante}})$$

also written informally as $c' = c + \Delta\Phi$.

Theorem (the 'potential theorem'). *The modified costs defined in this way are valid amortized costs, i.e. they satisfy the fundamental inequality of amortized analysis.*

This begs the question: how do we come up with potential functions? It's generally easier to come up with useful potential functions than it is to come up with amortized costs from scratch. We'll give some guidance in a moment, after looking at an example.

EXAMPLE ANALYSIS USING POTENTIAL FUNCTIONS

Consider the dynamic array from page 53, where the cost of writing an element is 1 and the cost of doubling and copying from capacity m to $2m$ is κm. Define the potential function

$$\Phi = \kappa \left[2 \left(\begin{array}{c} \text{num. items} \\ \text{in array} \end{array} \right) - \begin{array}{c} \text{capacity} \\ \text{of array} \end{array} \right]^+$$

Notation: $[x]^+$ means $\max(0, x)$.

This is clearly a valid potential function since it's ≥ 0 and $= 0$ at the initial empty state.

Let's run through a sequence of append operations, to get a feel for how this potential function behaves. We'll annotate each operation with its true cost c and its amortized cost $c + \Delta\Phi$.

initially empty
$\square$ $\Phi = 0$
append: $c = 1$, $c + \Delta\Phi = 1 + \kappa$
$\boxed{\bullet}$ $\Phi = \kappa$
append with doubling: $c = \kappa + 1$, $c + \Delta\Phi = 2\kappa + 1$
$\boxed{\bullet\bullet}$ $\Phi = 2\kappa$
append with doubling: $c = 2\kappa + 1$, $c + \Delta\Phi = 2\kappa + 1$
$\boxed{\bullet\bullet\bullet\ }$ $\Phi = 2\kappa$
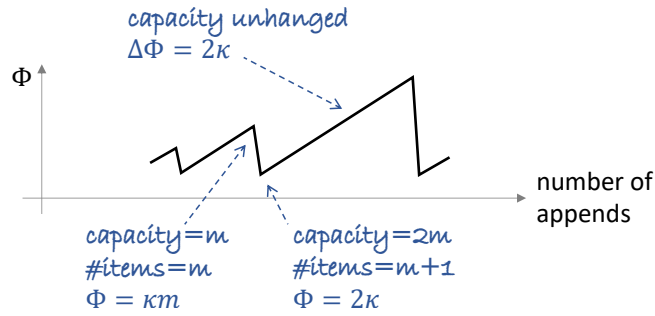append: $c = 1$, $c + \Delta\Phi = 2\kappa + 1$
$\boxed{\bullet\bullet\bullet\bullet}$ $\Phi = 4\kappa$
append with doubling: $c = 4\kappa + 1$, $c + \Delta\Phi = 2\kappa + 1$
$\boxed{\bullet\bullet\bullet\bullet\bullet\ \ \ }$ $\Phi = 2\kappa$

In our run-through, for each operation the amortized cost turned out to be $c + \Delta\Phi \leq 2\kappa + 1$. Now let's write out the full argument, to show that this holds in general. There are two ways that append could play out:

- It could be that the capacity needs to double from m to $2m$. The true cost of this type of append is $c = \kappa m + 1$. The change in potential is $\Delta\Phi = 2\kappa - \kappa m$. Thus the amortized cost is $c + \Delta\Phi = 2\kappa + 1$.

- Or it could be that we don't need to double capacity. The true cost of this type of append is $c = 1$. The change in potential is $\Delta\Phi = 2\kappa$. Again, the amortized cost is $c + \Delta\Phi = 2\kappa + 1$.

- There is only one corner case that we need to worry about, the very first append. For the initial empty state the number of items (zero) is less than half the capacity (one), and so the $[\cdot]^+$ in the definition of the potential function kicks in. For every other state, the number of items is $\geq$ half the capacity, and so the $[\cdot]^+$ doesn't kick in. We've already calculated the amortized cost of the very first append in the run-through above, and we found it to be $\kappa + 1$.

In all cases the amortized cost of an append is $\leq 2\kappa + 1 = O(1)$.

In practice, we'd usually write out the amortized analysis using sloppier notation:

Assume that the cost of append is $O(1)$ if there's no doubling needed, and $O(n)$ if we need to double an array with n items. Define the potential function $\Phi = 2n - c$ where n is the number of items in the array and c is the capacity. There are two ways that append could play out:

- *If there are n items and the capacity is n, we need to double the capacity. The true cost is $c = O(n)$ and the change in potential is $\Delta\Phi = [2(n+1) - 2n] - [2n - n] = 2 - n$, so the amortized cost is $O(n) + 2 - n = O(1)$.*

- *Otherwise we don't need to double the capacity. The true cost is $c = O(1)$ and the change in potential is $\Delta\Phi = 2$, so the amortized cost is $O(1) - 2 = O(1)$.*

Our Φ function isn't quite a potential function—we should have $\Phi \geq 0$ everywhere, and $\Phi = 0$ at the initial empty state, and this isn't the case. But we can special-case $\Phi(empty) = 0$, and this is a modification at just a single state so the asymptotic results are still valid. We conclude that in all cases the amortized cost of append is $O(1)$.
□

It's perfectly fine to use this sloppy way of writing, as long as you understand the real reasoning behind the otherwise preposterous equation "$c + \Delta\Phi = O(n) + 2 - n = O(1)$". What it really means is "The true cost is $\leq \kappa n$, for n sufficiently large, and the change in potential is $\Delta\Phi = 2 - n$. We should really have defined a different potential function, by multiplying by κ. If we had done that, then the amortized cost would be $\leq \kappa n + 2 - \kappa n = O(1)$."

DESIGNING POTENTIAL FUNCTIONS

Where do potential functions come from? We can invent whatever potential function we like, and different choices might lead to different amortized costs of operation. If we are cunning in our choice of potential function, we'll end up with informative amortized costs. There is no universal recipe. Here are some suggestions:

- Sometimes we want to say "this operation may technically have worst case $O(n)$ but morally speaking it should be $O(1)$", as with MinList.min on page 51 and with DynamicArray.append. To get amortized cost $O(1)$, the potential has to build up to n before the expensive operation, and drop to 0 after—or, to be precise, build up to $\Omega(n)$ and drop to $O(1)$. Think of Φ as storing up credit for the cleanup we're going to have to do.

- More generally, think of Φ as measuring the amount of stored-up mess. If an operation increases mess ($\Delta\Phi > 0$) then this mess will have to be cleaned up eventually, so we set the amortized cost to be larger than the true cost, to store up credit. An operation that does cleanup will decrease mess ($\Delta\Phi < 0$), which can cancel out the true cost of the cleanup operation.

- When you invent a potential function, make sure that $\Phi \geq 0$ and that $\Phi = 0$ for the initial empty data structure. Otherwise you'll end up with spurious amortized costs. This is a common source of errors! (See the example sheet.)

The goal of designing a potential function is to obtain useful amortized costs, and the goal of amortized analysis is to get tight bounds on the worst-case performance of a sequence of operations on a data structure. So, to figure out if our potential function is well-designed, we should see if it gives tight bounds.

For example, suppose we've used our potential function to prove that "operation popmin has amortized cost $O(\log N)$ for a data structure containing $\leq N$ items". This proves that the aggregate cost of m calls to popmin is[15] $O(m \log N)$. To check whether this bound is tight, we sould look for a concrete example of a sequence of m operations that has cost $\Omega(m \log N)$. Typically, m will grow with N. If we can find such a lower bound, we know that the amortized costs are tight. If on the other hand we had chosen a daft potential function, we'd end up with a Ω–O gap.

> **Exercise.**
> Consider the MinList data structure from page 51, implemented using the Stage 1 method. By designing a suitable potential function, show that append and min both have amortized cost $O(1)$.

---

*Each **append** creates 'mess' in the form of values that will need to be trawled through by the next call to **min**. So we want the potential to increase on each call to **append**, enough to 'pay for' the work that the next **min** will do. So let's define*

$$\Phi = num.\ items\ that\ \textbf{min}\ hasn't\ processed.$$

*Then the amortized cost of **append** is $c + \Delta\Phi = O(1) + 1 = O(1)$, and the amortized cost of **min** is $O(L) + (0 - L) = O(1)$, where L is the number of items that it has to process. Thus both of these operations have amortized cost $O(1)$.*
□

---

[15]This is an asymptotic statement in N, and m is allowed to depend on N. Remember that the fundamental inequality of amortized analysis, the inequality on page 53, is required to hold for *every* sequence of operations.

The whole analysis using potential functions rests on the 'potential theorem'. Let's restate it, and give a proof.

Theorem (the 'potential theorem'). *Let Φ be a potential function. For an operation that takes the state from $\mathcal{S}_{ante}$ to $\mathcal{S}_{post}$, and that has true cost c, define the modified cost to be*

$$c' = c + \Phi(\mathcal{S}_{post}) - \Phi(\mathcal{S}_{ante}).$$

The modified costs defined in this way are valid amortized costs. In other words they satisfy the fundamental inequality of amortized analysis: for any sequence of operations

$$\begin{array}{cc} \textit{aggregate true cost} \\ \textit{of the sequence} \end{array} \leq \begin{array}{c} \textit{aggregate amortized cost} \\ \textit{of that sequence.} \end{array}$$

Proof. Consider a sequence of operations starting from an initially empty state,

$$\mathcal{S}_0 \xrightarrow{c_1} \mathcal{S}_1 \xrightarrow{c_2} \mathcal{S}_2 \xrightarrow{c_3} \cdots \xrightarrow{c_k} \mathcal{S}_k$$

where the true costs are $c_1, c_2, \ldots, c_k$. Then

This 'telescoping sum' trick also appeared in the analysis of Johnson's algorithm, section 5.8 page 26

aggregate amortized cost

$$= \left\{ -\Phi(\mathcal{S}_0) + c_1 + \Phi(\mathcal{S}_1) \right\} + \left\{ -\Phi(\mathcal{S}_1) + c_2 + \Phi(\mathcal{S}_2) \right\}$$

$$+ \cdots + \left\{ -\Phi(\mathcal{S}_{k-1}) + c_k + \Phi(\mathcal{S}_k) \right\}$$

$$= c_1 + \cdots + c_k - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k)$$

$$= \text{aggregate true cost} - \Phi(\mathcal{S}_0) + \Phi(\mathcal{S}_k)$$

hence

aggregate true cost

$$= \text{aggregate amortized cost} + \Phi(\mathcal{S}_0) - \Phi(\mathcal{S}_k)$$

$$\leq \text{aggregate amortized cost} + \Phi(\mathcal{S}_0) \quad (\text{since } \Phi \geq 0)$$

$$= \text{aggregate amortized cost} \quad (\text{since } \Phi = 0 \text{ for the initial empty state } \mathcal{S}_0).$$

Thus the costs c' that we defined are indeed valid amortized costs. $\square$

7.5. Three priority queues

All of the work in this section is a build-up to a very advanced implementation of the Priority Queue, called the Fibonacci Heap.

```
AbstractDataType PriorityQueue:
    # Holds a dynamic collection of items.
    # Each item has a value/payload v, and a key/priority k.

    # Extract the item with the smallest key
    Pair<Key, Value> popmin()

    # Add v to the queue, and give it key k
    push(Value v, Key k)

    # For a value already in the queue, give it a new (lower) key
    decreasekey(Value v, Key newk)

    # Sometimes we also include methods for:
    # merge two priority queues
    # delete a value
    # peek at the item with smallest key, without removing it
```

It's useful to review the two implementations that we've already seen. We'll also look at a third very simple implementation, the linked-list implementation, as a thought experiment to sharpen our thinking and to highlight where there's room for improvement. This table summarizes the running for those three implementations, as well as for the Fibonacci Heap. In this table N is the maximum number of items in the heap.[16]

	popmin	push	decreasekey
binary heap	$O(\log N)$	$O(\log N)$	$O(\log N)$
binomial heap	$O(\log N)$	$O(1)$ amortized	$O(\log N)$
linked list	$O(N)$	$O(1)$	$O(1)$
Fibonacci heap	$O(\log N)$ amortized	$O(1)$ amortized	$O(1)$ amortized

BINARY HEAP*

*This section of notes is a recap of section 4.8.1. A *binary heap* is an almost-full binary tree (i.e. every level except the bottom is full), so its height is $\lfloor \log_2 n \rfloor$ where n is the number of elements. It satisfies the heap property (each node's key is $\leq$ its children), so the minimum of the entire heap can be found at the root.

Notation: $\lfloor x \rfloor$ is the floor of x, i.e. $\lfloor x \rfloor \leq x < \lfloor x \rfloor + 1$.
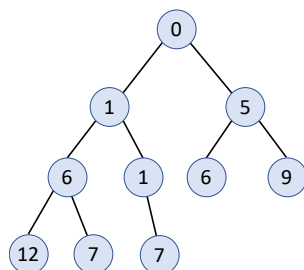

The heap property:
each node's key is $\leq$ those
of its children

To implement popmin we extract the root item, replace the root by the end element, and then bubble it far enough down so as to satisfy the heap property. The number of bubble-down steps is limited by the height of the tree, so popmin is $O(\log n)$.

---

[16]For naive worst-case analysis, N is simply the number of items in the heap when we do the operation. But amortized analysis applies to sequences of operations, and the number of items will fluctuate over those operations, and so we have to instead define N as "upper bound on the number of items in the heap over the sequence of operations in question".
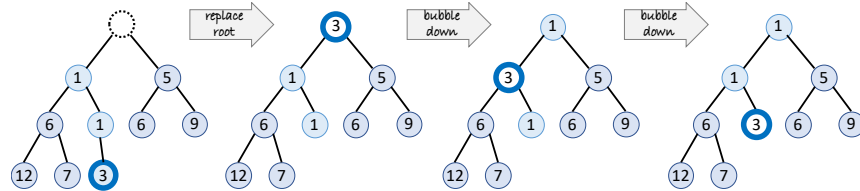
To implement push we append the new item to the very end, and then bubble it far enough up the tree so as to satisfy the heap property. Again, the number of bubble-up steps is $O(\log n)$. And decreasekey is very similar.
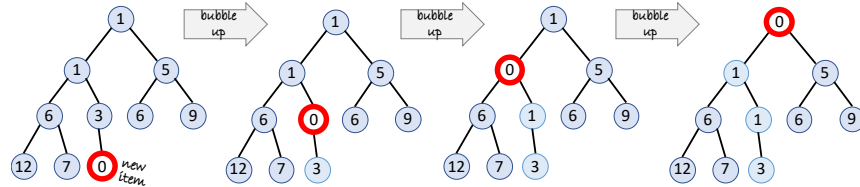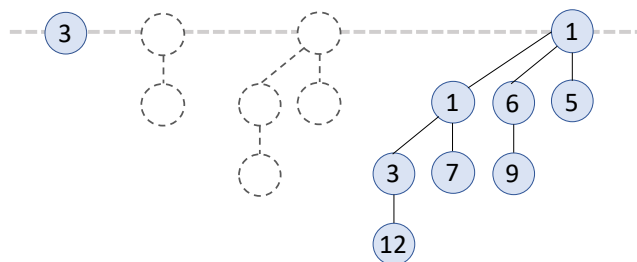

BINOMIAL HEAP*

*This section of notes is a recap of section 4.8.2. A *binomial tree of degree 0* is a single node. A *binomial tree of degree k* is a tree obtained by combining two binomial trees of degree $k - 1$, by appending one of the trees to the root of the other.
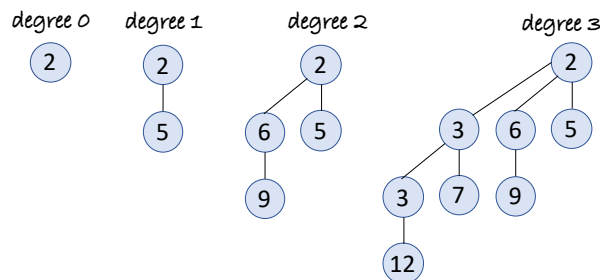
A *binomial heap* is a collection of binomial trees, at most one for each tree degree, each obeying the heap property i.e. each node's key is $\leq$ those of its children. Here is a binomial heap consisting of one binomial tree of degree 0, and one of degree 3. (The dotted parts in the middle indicate 'there is no tree of degree 1 or 2'.)

Here are some basic properties of binomial trees and heaps.

1. A binomial tree of degree k has 2^k nodes
2. A binomial tree of degree k has height k
3. In a binomial tree of degree k, the root node has k children (which is why we call it 'degree')
4. In a binomial tree of degree k, the root node's k children are binomial trees of all the degrees $k - 1$, $k - 2$, ..., 0.
5. In a binomial heap with n nodes, the 1s in the binary expansion of the number n correspond to the degrees of trees contained in the heap. For example, a heap with 9 nodes (binary $1001 = 2^3 + 2^0$) has one tree of degree 3 and one tree of degree 0.
6. If a binomial heap contains n nodes, it contains $O(\log n)$ binomial trees, and the largest of those trees has degree $O(\log n)$.

The operations on binomial heaps end up resembling binary arithmetic, thanks to property 5.

push(v, k) is $O(\log n)$:
>   Treat the new item as a binomial heap with only one node, and merge it as described
>   below, at cost $O(\log n)$, where n is the total number of nodes. It can be shown that
>   the amortized cost is $O(1)$ — see the example sheet.

decreasekey(v, newk) is $O(\log n)$:
>   Proceed as with a normal binary heap, applied to the tree to which v belongs. The
>   entire heap has $O(n)$ nodes, so this tree has $O(n)$ nodes and height $O(\log n)$, so the
>   cost of decreasekey is $O(\log n)$.

popmin() is $O(\log n)$:
>   First scan the roots of all the trees in the heap, at cost $O(\log n)$ since there are that
>   many trees, to find which root to remove. Cut it out from its tree. Its children form a
>   binomial heap, by property 4. Merge this heap with what remains of the original one,
>   as described below, at cost $O(\log n)$.

merge(h1, h2) is $O(\log n)$:
>   To merge two binomial heaps, start from degree 0 and go up, as if doing binary addition,
>   but instead of adding digits in place k we merge binomial trees of degree k, keeping the
>   tree with smaller root on top. If n is the total number of nodes in both heaps together,
>   then there are $O(\log n)$ trees in each heap, and $O(\log n)$ operations in total.

LINKED LIST PRIORITY QUEUE

Here's a very simple priority queue. It uses a doubly-linked list to store all the items, and it
also keeps a pointer to the smallest item.
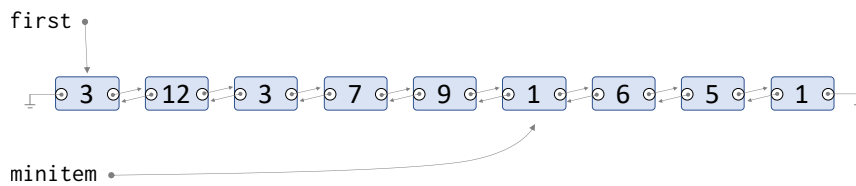
push(v, k) is $O(1)$:
>   just attach the new item to the front of the list, and if k < minitem.key then update
>   minitem

decreasekey(v, newk) is also $O(1)$:
>   update v's key, and if newk < minitem.key then update minitem

popmin() is $O(n)$:
>   we can remove minitem in $O(1)$ time, but to find the new minitem we have to traverse
>   the entire list.

7.6. Fibonacci heap

The Fibonacci heap is a fast priority queue. It was developed by Fredman and Tarjan in 1984, specifically to speed up Dijkstra's algorithm.

GENERAL IDEA

Here is an outline of the thinking that led to the Fibonacci heap.

1. On a graph with V vertices and E edges, Dijkstra's algorithm might make V calls to popmin, and E calls to push and/or decreasekey. Since E might be as big as $\Omega(V^2)$, push and decreasekey are the common operations, so we want them to be $O(1)$.

2. To make push and decreasekey be $O(1)$, they have to be lazy and they should only 'touch' a small part of the data structure. The linked list implementation shows us one way we can make push be lazy — just dump new nodes into a list, with no further tidying. Nor should decreasekey do any tidying — if decreasing a key leads to a heap violation, then the offending node should just be dumped into a list, with no further tidying.

   The binomial heap actually uses a 'binary counter' structure so that, most of the time, push only needs to touch a few small trees, and this gives amortized cost $O(1)$. The Fibonacci heap uses the same trick, but inside popmin rather than push.

3. In a heap, a call to popmin extracts the root of a tree, and so all of its children need to be processed. Thus popmin has complexity $\Omega(d)$ where d is the number of children, also called the *degree*.

4. We don't want our heap to have any wide shallow trees — they would require popmin to do a lot of work, just as bad as the simple linked list implementation. We need to limit the degree of a node in the heap. We need a mechanism to ensure that the trees are deep and bushy, as they are in a binomial tree.

5. We shall from time to time do expensive housekeeping. If we're accumulating mess that will have to be cleaned up anyway, why not just clean up as we go? The heart of the answer lies in our analysis of heapsort in Section 2.10. We saw that it takes time $O(n \log n)$ to add n items to a binary heap one by one, but only $O(n)$ to heapify them in a batch.

   Doing housekeeping in batches is the big idea behind the Fibonacci heap. The housekeeping is split between popmin and decreasekey. We will invent a potential function to help us reason about the amount of housekeeping.

HOW TO PUSH AND POPMIN

The Fibonacci heap, like the binomial heap, stores a list of heaps. Unlike the binomial heap, the trees can have any shape. Like the linked list priority queue, we'll keep track of minroot, the smallest element in the data structure, which must of course be the root of one of the heap.
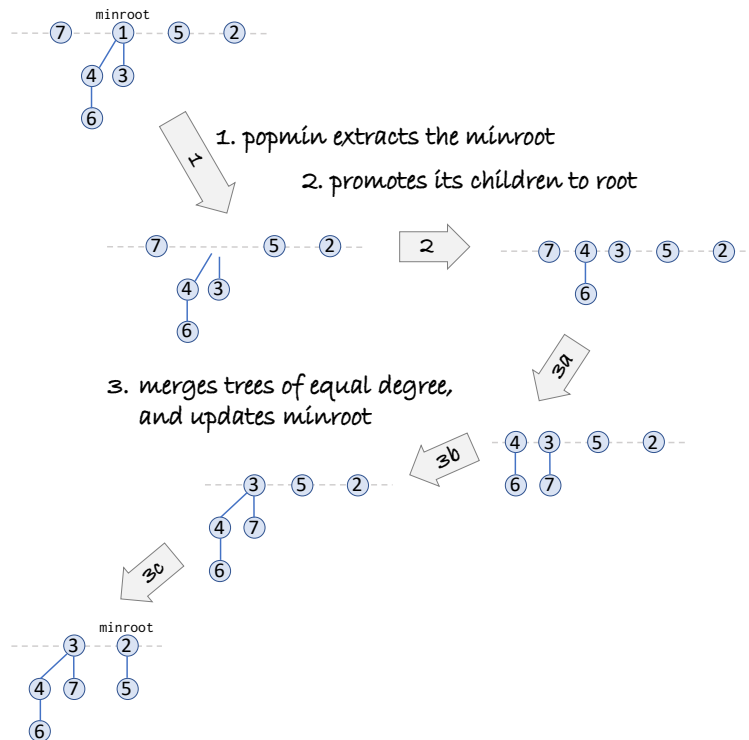
```
1   # Maintain a list of heaps (i.e. store a pointer to the root of each heap)
2   roots = []
3
4   # Maintain a pointer to the smallest root
5   minroot = None
6
7   def push(v, k):
8       create a new heap h consisting of a single item (v, k)
9       add h to the list of roots
10      update minroot if k < minroot.key
11
```

```
12  def popmin():
13      take note of minroot.value and minroot.key
14      delete the minroot node, and promote its children to be roots
15      # cleanup the roots
16      while there are two roots with the same degree:
17          merge those two roots, by making the larger root a child of the smaller
18      update minroot to point to the smallest root
19      return the value and key from line 13
```


1. popmin extracts the minroot

2. promotes its children to root

3. merges trees of equal degree, and updates minroot

In this simple version, with only push and popmin, one can show by induction that the Fibonacci heap consists at all times of a collection of binomial trees, and that after the cleanup in lines 16–17 it is a binomial heap. (See the example sheet.)

It doesn't matter how the cleanup is implemented, as long as it is done efficiently. Here is an example implementation.

```
20  def cleanup(roots):
21      root_array = [None, None, ...]   # empty array
22      for each tree t in roots:
23          x = t
24          while root_array[x.degree] is not None:
25              u = root_array[x.degree]
26              root_array[x.degree] = None
27              x = merge(x, u)
28          root_array[x.degree] = x
29      return list of non–None values in root_array
```

HOW TO DECREASEKEY

If we can decrease the key of an item in-place (i.e. if its parent is still $\leq$ the new key), then that's all that decreasekey needs to do. If however the node's new key is smaller than its parent, we need to do something to maintain the heap. We've already discussed why it's a reasonable idea to be lazy—to just cut such a node out of its tree and dump it into the root list, to be cleaned up in the next call to popmin.

There is however one extra twist. If we just cut out nodes and dump them in the root list, we might end up with trees that are shallow and wide, even as big as $\Omega(n)$, where n is the number of items in the heap. This would make popmin very costly, since it has to iterate through all minroot's children.
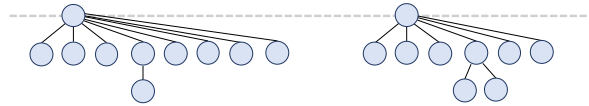
To make popmin reasonably fast, we need to keep the maximum degree small. The Fibonacci heap achieves this via two rules:

1. Lose one child, and you get marked as a 'loser' node.
2. Lose two children, and you get dumped into the root list (and your mark is removed).

This ensures that the trees end up with a good number of descendants. Formally, we'll show in Section 7.8 that a tree with degree d contains $\geq 1.618^d$ nodes, and hence that the maximum degree in a heap of n items is $O(\log n)$.
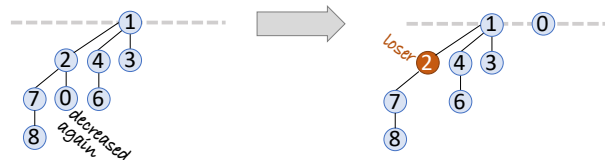
Similarly, a binomial tree of degree k has 2^k nodes, which implies a binomial heap of n items has maximum degree $O(\log n)$.
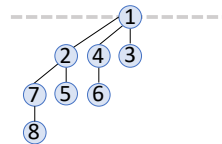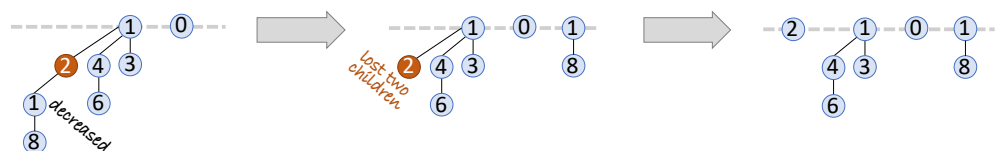

decreasekey from 5 to 3
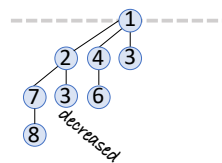
decreasekey again to 0— move 0 to maintain the heap

decreasekey from 7 to 1— move 1 to maintain the heap— move the double-loser to root

```
30  # Every node will store a flag, p.loser = True / False
31
32  def decreasekey(v, k′):
33      let n be the node where this value is stored
34      n.key = k′
35      if n is a root:
36          update minroot if necessary
37      else, if n violates the heap condition:
38          repeat:
39              p = n.parent
40              remove n from p.children
41              insert n into the list of roots, updating minroot if necessary
42              n.loser = False
43              n = p
44          until p.loser == False
45          if p is not a root:
46              p.loser = True
47
48  def popmin():
49      mark all of minroot's children as loser = False
50      then do the same as in the simple version, lines 13–19
```

Here is another example of the operation of decreasekey, this time highlighting what happens if there are multiple loser ancestors.
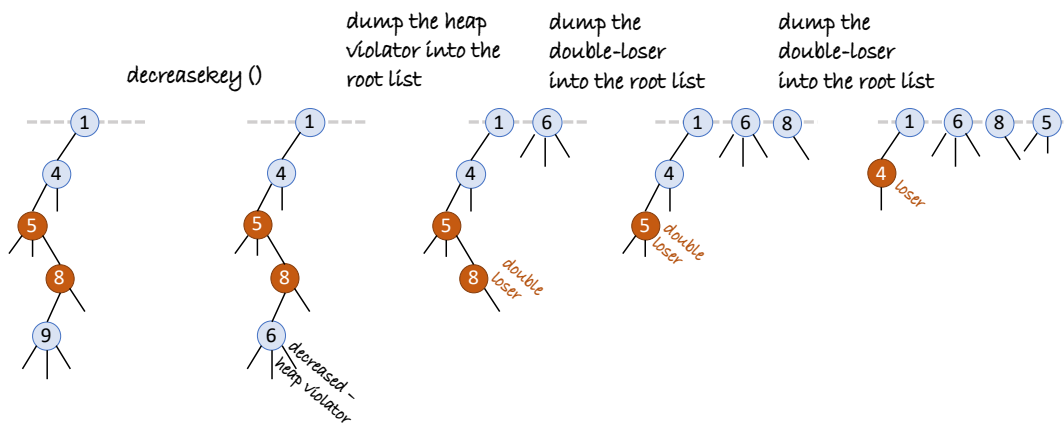
7.7. Implementing the Fibonacci heap*

* This section of notes is not examinable.

There's a question that gets asked every year when students learn about the Fibonacci heap:
"How do we find the node that we want to call decreasekey on? Do we trawl through the
entire heap? Isn't this $O(N)$?"

The mental model behind this question is as follows. Suppose we have a Fibonacci
heap in which each node has a pointer to a parent node (except for root nodes which don't
have parents) as well as to child nodes; and suppose each node also contains a payload, for
example an object representing a graph vertex. If we're given a graph vertex w and we want
to call decreasekey we'd have to first find the Fibonacci heap node that contains it, and then
perhaps rewire the Fibonacci heap's parent/child pointers.

```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)
    while not toexplore.isempty():
        v = toexplore.popmin()
        for (w,edgecost) in v.neighbours:
            ...
            # we may do one of these two:
                toexplore.push(w)
                toexplore.decreasekey(w)
```
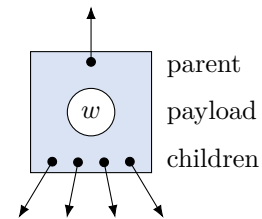

There's something fishy about this question. There are two competing worldviews here—one
view says that vertices are nodes in a Fibonacci heap, the other that they are objects in a
graph—and why should one worldview take precedence over the other?[17]
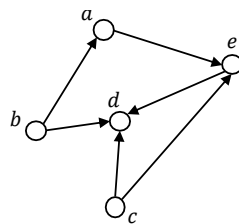
vertices in a graph;
arrows show
graph-neighbours
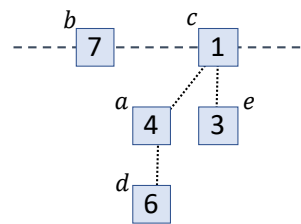
the same vertices;
edges show
heap-relationships

Here's a more thoughtful way to formulate the problem: When we use the Fibonacci heap
as the priority queue in Dijkstra's algorithm, each vertex is doing double duty: it is at once
both a vertex in the graph and also a node in the Fibonacci heap. They are participating in
two data structures *simultaneously*. How should we implement this?

JAVANILE SOLUTION

The naive solution is to just throw everything into a single class, as in the code below. Every
VertexNode object does double duty. We don't need to look up the Fibonacci heap node that
contains a graph vertex w, because the node *is* the vertex.

```
class VertexNode:
    # used by Dijkstra:
    List<VertexNode> graph_neighbours
    float distance
```

----

[17]In the five years I've been teaching this material, every year students have asked "how do you find the
node you want to call decreasekey on?". But not once have they asked the question the other way round—not
once have they started from the mental model "each vertex object contains a payload that is a node object"
and asked the obvious question "When we call popmin, and find the minroot node, how do we find the vertex
object that contains it? Do we trawl through the entire graph? Isn't this $O(V)$?"

```
    VertexNode come_from

    # used by FibHeap:
    VertexNode fib_parent
    List<VertexNode> fib_children
    float key
    bool is_loser
```

This is an ugly solution because it ties the two data structures together. All the library code for the Fibonacci heap is tied to this particular class, and so it isn't reusable by other programmers who aren't interested in graphs and just want their own priority queue. Can we tease the two data structures apart without paying a price?

Here's an implementation for programmers who haven't outgrown an infatuation with classes and templates. We'll let Vertex represent a vertex in the graph, Node represent a node in the Fibonacci heap, and we'll declare that each Vertex *uses* a Node to store its heap-related pointers.
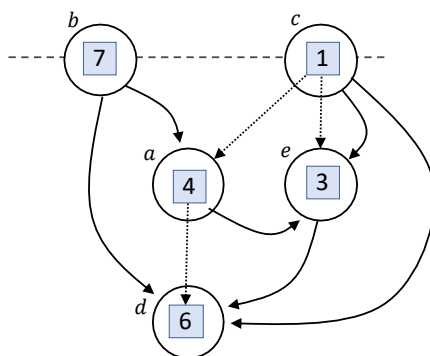
The heap-related pointers stored by Node have to point to Vertex objects, rather than to Node objects, since otherwise we're stuck with the problem "how do I find the Vertex object for a given Node?" But we want Node to be general purpose, not tied to a graph. The solution is to use interfaces and templates, to allow the Fibonacci heap routines to 'see through' a Vertex to get to its Node without needing to know anything about how Vertex is implemented.

```
class FibHeap<T extends FibHeap.Nodeable<T>>:
    class Node<T>:
        float key
        int degree
        T parent
        List<T> children
    interface Nodeable<T>:
        Node<T> get_fib_node()

    List<T> roots
    def T popmin(): ...
    def push(T value, float key): ...
    def decrease_key(T value, float newkey): ...

class Graph:
    class Vertex implements FibHeap.Nodeable<Vertex>:
        List<Pair<Vertex, float>> neighbours
        float distance
        Vertex? come_from
        FibHeap.Node<Vertex> pqn
        FibHeap.Node<Vertex> get_fib_node(): return pqn
    def compute_shortest_paths_from(Vertex s):
        ...
```

DYNAMIC ARCHITECTURE

A cleaner approach is to label each vertex by an id, and to use two hash tables: one to look up a vertex given its id, and another to look up a heap node given the id. (Often there is a natural id to use for each vertex, for example, the node id in an OpenStreetMap graph, or the primary key of the graph as stored in a database.) The interfaces for both the Fibonacci heap and the graph are defined in terms of ids.
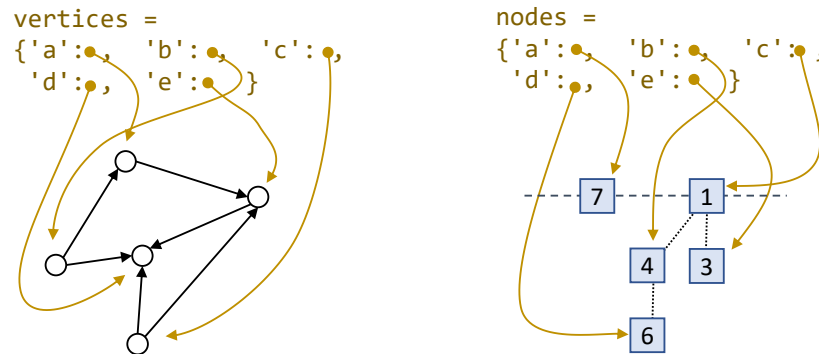
The advantage of this architecture is that it keeps the code for graph traversal entirely separate from that for the priority queue. This would be especially suitable if we start with a graph and we don't know ahead of time which algorithms we'll need to run on it; in this scenario it doesn't make sense to have a Vertex class that embodies a particular storage requirement.

```
class FibHeap:
    HashMap <Id , FibHeapNode> nodes
    class FibHeapNode:
        Id id
        float key
        int degree
        FibHeapNode parent
        List<FibHeapNode> children

    List<FibHeapNode> roots
    def Id popmin(): ...
    def push(Id node, float key): ...
    def decrease_key(Id node, float newkey): ...

class Graph:
    HashMap <Id , Vertex> vertices
    class Vertex:
        Id id
        List<Pair<Vertex , float>> neighbours
        float distance
        Vertex? come_from
    def compute_shortest_paths_from(Id s):
        ...
```

NITTY GRITTY

It's worth thinking in a little more detail how exactly to store the parent and children pointers of each node in the Fibonacci heap. We have to perform various slicing and rearranging operations on these pointers—and it would be silly to put in lots of effort designing a very clever amortized design and then waste it all with an inefficient implementation! The manipulations we want to perform are:
- slice a node out of a tree in $O(1)$
- add a node to the root list in $O(1)$
- merge two trees in $O(1)$
- iterate through a root's children in $O(\text{num. children})$

These can all be achieved by keeping enough pointers around. We can use a circular doubly-linked list for the root list; and the same for each list of siblings; and we'll let each node point to its parent; and each parent will point to one of its children. So, for example, to iterate through a node's children, we first follow the down-pointer to get to one of the children, then we follow the sibling points around until they bring us back to the initial child.
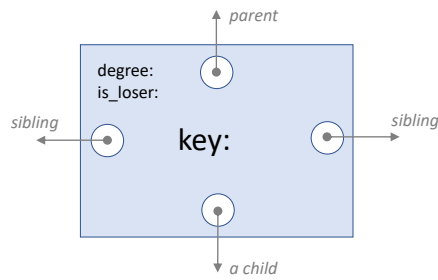
So this little Fibonacci heap
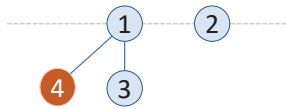
would be represented as
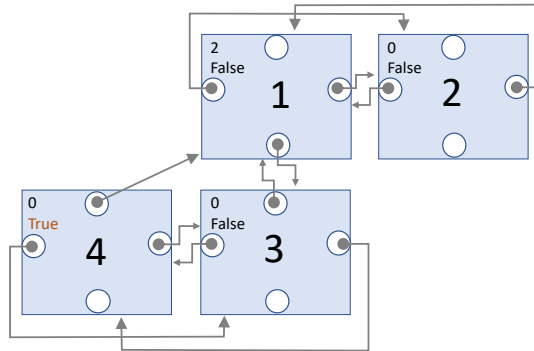
7.8. Analysis of Fibonacci heap

We'll now compute the amortized costs of the various operations on the Fibonacci heap, using the potential function

$$\Phi = \text{number of roots} + 2\big(\text{number of loser nodes}\big).$$

Let n be the number of items in the heap, and let $d_{\max}$ be an upper bound on the degree of any node in a Fibonacci heap (we'll see soon that $d_{\max} = O(\log n)$ is suitable).

push() : amortized cost $O(1)$
>   This just adds a new node to the root list, so the true cost is $O(1)$. The change in potential is $\Delta\Phi = 1$, so the amortized cost is $O(1)$.

popmin() : amortized cost $O(d_{\max})$
>   Let's split this into three parts. First, cut out minroot and promote its children to the root list. There are at most $d_{\max}$ children to promote, so the true cost is $O(d_{\max})$. These children get promoted to root, and maybe some of them lose the loser mark, so $\Delta\Phi \leq d_{\max}$. So the amortized cost for this first part is $O(d_{\max})$.
>
>   The second part is running cleanup. Line 21 initializes an array, in which we will store a tree of degree d in root_array[d]. We defined $d_{\max}$ to be the largest possible degree in a Fibonacci heap with n items, so an array of size $d_{\max} + 1$ is sufficient,[18] so it takes $O(d_{\max})$ to initialize the array. Now, suppose that cleanup starts with x trees, it does M merges, and ends up with y trees. The total true cost of this is
>
> $$\begin{aligned}
\text{cost} &= O(x + M + y) \\
&\quad \text{for processing each tree, lines 22–23 \& 28: } O(x) \\
&\quad \text{for the merges, lines 24–27: } O(M) \\
&\quad \text{for the final copy out of root\_array, line 29: O(y)} \\
&= O(y + 2M + d_{\max} + 1) \\
&\quad \text{every merge decreases the number of trees by 1 so } x - M = y \\
&= O(d_{\max} + 1 + 2M + d_{\max} + 1) \\
&\quad \text{we end up with at most one tree per cell of the array so } y \leq d_{\max} + 1 \\
&= O(M + d_{\max}).
\end{aligned}$$
>
>   And $\Delta\Phi = -M$ because our M merges decreased the number of roots by M. Thus the amortized cost is $O(M - d_{\max}) - M = O(d_{\max})$.
>
>   The third and final part is fixing up minroot, line 18. This can be done by scanning through the list of trees, which is $O(d_{\max})$.

decreasekey() : amortized cost $O(1)$
>   It takes $O(1)$ elementary operations to decrease the key. If the node doesn't have to move, then Φ doesn't change, so amortized cost = true cost = $O(1)$. If the node does have to move, the following happens:
>
>   - We move the node, call it a, to the root list. The true cost is $O(1)$, and Φ increases by ≤ 1: it increases by 1 if a wasn't a loser, and decreases by 1 if it was.
>   - Some of a's loser ancestors b and c are moved to the root list. Say there are L of these. The true cost of moving is $O(L)$. And $\Delta\Phi = L - 2L$: the L is because of the new roots, and the $-2L$ is because their loser marks are erased. Thus the amortized cost of this step is zero, regardless of L.
>   - One ancestor d might have to be marked as a loser. The true cost is $O(1)$, and Φ increases by 2, so the amortized cost is $O(1)$.

We can see now why the potential function was chosen just so. The two operations popmin and decreasekey both include an uncontrolled number of steps, M for popmin and L for decreasekey. But these steps have already been 'paid for', by the operations that increased Φ, so they contribute zero to the amortized cost.

---
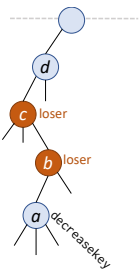
[18]A common question: 'Doesn't it need to be size $d_{\max} + 2$, in case there were two trees of degree $d_{\max}$ before the cleanup, resulting in a tree of degree $d_{\max} + 1$ after the cleanup?' No. We defined $d_{\max}$ to be the *maximum possible* degree in the Fibonacci heap, so the maximum degree even after cleanup is by definition $\leq d_{\max}$.

BOUNDING THE SHAPE OF THE TREES

The amortized cost of popmin is $O(d_{\max})$, where $d_{\max}$ is the maximum number of children of any of the nodes in the heap. The peculiar mechanism of decreasekey was designed to keep $d_{\max}$ small. How small?

Theorem (Fibonacci shape theorem). *If a node in a Fibonacci heap has d children, then the subtree rooted at that node has $\geq F_{d+2}$ nodes, where $F_1, F_2, \ldots$ are the Fibonacci numbers.*

$F_1 = 1,\ F_2 = 1,$
$F_3 = 2,\ F_4 = 3,$
$F_5 = 5,\ \ldots$ The
general formula for
F_n is
$(\phi^n - (-\phi)^{-n})/\sqrt{5}.$

It's a mathematical fact from linear algebra that $F_{d+2} \geq \phi^d$ where ϕ is the golden ratio, $\phi = (1 + \sqrt{5})/2 \approx 1.618$. It's a simple exercise (left to the example sheet) to deduce from this fact and the theorem that $d_{\max} = O(\log n)$.
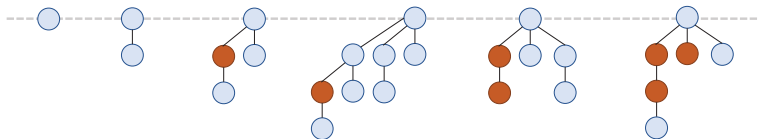
Proof (Fibonacci shape theorem). Consider an arbitrary node x in a Fibonacci heap, at some point in execution, and suppose it has d children, call them $y_1, \ldots, y_d$ in the order of when they last became children of x. (There may be other children that x acquired then lost in the meantime, but we're not including those.)
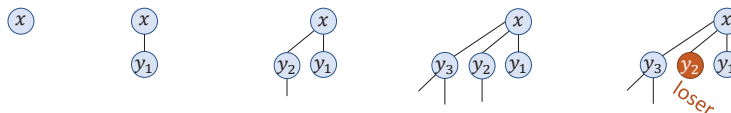
When x acquired y_2, x already had y_1 as a child, so y_2 must have had ≥ 1 child seeing as it got merged into x. Similarly, when x acquired y_3, y_3 must have had ≥ 2 children, and so on. After x acquired a child y_i, that child might have lost a child, but it can't have lost more because of the rules of decreasekey. Thus, at the point of execution at which we're inspecting x,

$$y_1 \text{ has } \geq 0 \text{ children}$$
$$y_2 \text{ has } \geq 0 \text{ children}$$
$$y_3 \text{ has } \geq 1 \text{ child}, \ldots$$
$$y_d \text{ has } \geq d - 2 \text{ children}.$$

Now for some pure maths. Consider an arbitrary tree all of whose nodes obey the grandchild rule "a node with children $i = 1, \ldots, d$ has at least $i - 2$ grandchildren via child i". Let N_d be the smallest possible number of nodes in a subtree whose root has d children. Then

$$N_d = \underbrace{N_{d-2}}_{\text{child } d} + \underbrace{N_{d-3}}_{\text{child } d-1} + \cdots + \underbrace{N_0}_{\text{child } 2} + \underbrace{N_0}_{\text{child } 1} + \underbrace{1}_{\text{the root.}}$$

Substituting in N_{d-1}, we get $N_d = N_{d-2} + N_{d-1}$, the defining equation for the Fibonacci sequence, hence $N_d = F_{d+2}$.

We've shown that the nodes in a Fibonacci heap obey the grandchild rule, therefore the number of nodes in the subtree rooted at x is $\geq F_{d+2}$ where d is the number of children of x. $\qquad\square$

7.9. Disjoint sets

The DisjointSet data structure (also known as union-find or merge-find) is used to keep track of a dynamic collection of items in disjoint sets. We used it in Kruskal's algorithm for finding a minimum spanning tree: each vertex of the graph corresponds to an item in the collection, and we used sets to track which vertices we had joined together into forest fragments.

In this section we'll first look three ways to implement this data structure. These implementations are interesting in their own right—but what's more interesting is to see the them as embodiments of the same sort of design strategies that led to the Fibonacci heap; and so we'll conclude the section by drawing out the analogy.

First, here's the specification of the DisjointSet abstract data type.

```
AbstractDataType DisjointSet:
    # Holds a dynamic collection of disjoint sets

    # Return a handle to the set containing an item.
    # The handle must be stable, as long as the DisjointSet is not modified.
    Handle get_set_with(Item x)

    # Add a new set consisting of a single item (assuming it's not been added already)
    add_singleton(Item x)

    # Merge two sets into one
    merge(Handle x, Handle y)
```
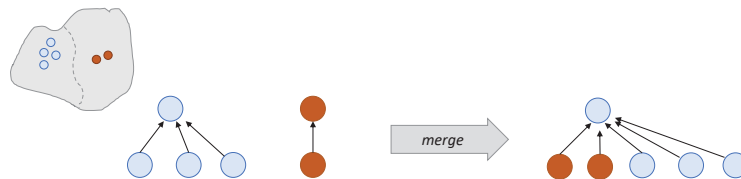
(This specification refer to Handles. It doesn't say what a handle *is*, only that handles don't change unless the DisjointSet is modified (by either add_singleton or merge). In practice, we might use a representative element from each set as the set's handle.)
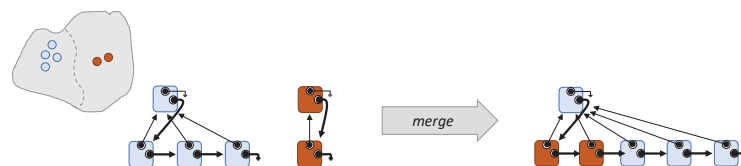
IMPLEMENTATION 1: FLAT FOREST

To make get_set_with fast, we could make each item point to its set's handle.

get_set_with() is just a single lookup.

merge() needs to iterate through each item in one or other set, and update its pointer. This takes $O(n)$ time, where n is the number of items in the DisjointSet.

To be able to iterate through the items, we could store each set as a linked list:

A smarter way to merge is to keep track of the size of each set, and pick the smaller set to update. This is called the *weighted union* heuristic. In the Example Sheet you'll show that the aggregate cost of any sequence of m operations on $\leq N$ elements (i.e. m operations of which $\leq N$ are add_singleton) is $O(m + N \log N)$, asymptotic in N.

IMPLEMENTATION 2: DEEP FOREST
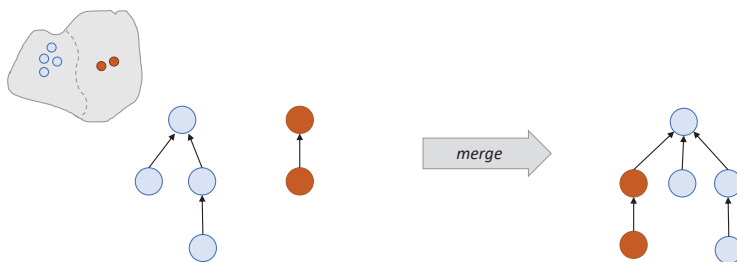
To make merge faster, we could skip all the work of updating the items in a set, and just build a deeper tree.

merge() attaches one root to the other, which only requires updating a single pointer.

get_set_with() needs to walk up the tree to find the root. This takes $O(h)$ time, where h is the height of the tree.

To keep h small, we can use the same idea as for the flat forest: keep track of the rank of each root (i.e. the height of its tree), and always attach the lower-rank root to the higher-rank. If the two roots had ranks r_1 and r_2 then the resulting rank is $\max(r_1, r_2)$ if $r_1 \neq r_2$, and $r_1 + 1$ if $r_1 = r_2$. This is called the *union by rank* heuristic. It can be shown that the aggregate cost of any sequence of m operations on $\leq N$ elements is $O(m \log N)$, asymptotic in N.

IMPLEMENTATION 3: LAZY FOREST

We'd like the forest to be flat so that get_set_with is fast, but we'd like to let it get deep so that merge can be fast. Here's a way to get the best of both worlds, inspired by the Fibonacci heap—defer cleanup until you actually need the answer.
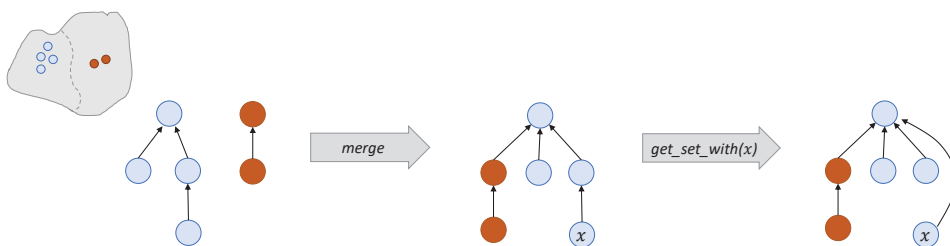
merge() is as for the deep forest.

get_set_with(x) does some cleanup. It walks up the tree once to find the root, and then it walks up the tree a second time to make x and all the intermediate nodes be direct children of the root.

This method is called the *path compression* heuristic. We won't adjust the stored ranks during path compression, and so rank won't be the exact height of the tree, just an upper bound on the height. (If we wanted to know the actual tree height we'd have to compute it—and we don't want the heuristic to take more time on book-keeping than it saves on actually doing the work!)

It can be shown that with the lazy forest the cost of m operations on $\leq N$ items is $O(m\alpha_N)$ where α_N is an integer-valued monotonically increasing sequence, related to the Ackerman function, which grows extremely slowly:

$\alpha_N = 0$ for $N = 0, 1, 2$

$\alpha_N = 1$ for $N = 3$

$\alpha_N = 2$ for $N = 4, 5, 6, 7$

$\alpha_N = 3$ for $8 \leq N \leq 2047$

$\alpha_N = 4$ for $2048 \leq N \leq 10^{80}$, more than there are atoms in the observable universe.

For practical purposes, α_N may be ignored in the O notation, and therefore the amortized cost per operation is $O(1)$.

LESSONS FOR DESIGNING ADVANCED DATA STRUCTURES

The three implementations of the DisjointSet can be thought of as embodying three different design strategies: anal retentive, adrenaline junkie, and forward planner. Those three strategies can also be seen in the implementations of the Priority Queue.[19]

Anal retentive Keeps everything pristine, all the time. Wastes effort by doing so.	**Binary heap** • push: slow $O(\log N)$, tidies up every time • popmin: fast $O(\log N)$, since heap is always tidy	**Flat forest** • merge: slow, rewrites parent points • get_set_with: fast $O(1)$, just look up parent
Adrenaline junkie Lets work pile up then does it all in a rush, and is too harried to learn from it.	**Linked list heap** • push: fast and lazy $O(1)$ • popmin: slow $O(N)$, does its work, but retains none of its findings for next time	**Deep forest** • merge: fast $O(1)$, updates a single pointer • get_set_with: slow, has to walk up the tree
Forward planner Does work when it's needed, and *keeps its working* so things are easier next time	**Fibonacci heap** • push: fast and lazy $O(1)$ • popmin: fast $O(\log N)$, leaves the heap semi-tidy for next time	**Lazy forest** • merge: fast and lazy $O(1)$ • get_set_with: leaves tracks that future operations can benefit from

The forward planner strategy means keeping your working, or at least some of it, in the anticipation of later operations. Here's another much simpler illustration of the general idea.

Example 7.2. Suppose we have a list of four objects

$$
\begin{array}{cccc}
a & b & c & d \\
\boxed{\text{val} = 2} & \boxed{\text{val} = 5} & \boxed{\text{val} = 3} & \boxed{\text{val} = 0}
\end{array}
$$

and we want to sort them by value, lowest value first. Suppose we run insertion sort:
- First, find the smallest. We need to make some comparisons. We'll find that a.val $<$ b.val, then a.val $<$ c.val, then a.val $>$ d.val, and conclude that d is the smallest.
- Next, find the second-smallest. We'll end up repeating some of the comparisons we've already made: we'll find a.val $<$ b.val, then a.val $<$ c.val, and conclude that a is the second smallest.

But this is a waste! We've already made those comparisons. We should have found a way to retain our findings from the first pass, for example by using a heap, so we don't duplicate the effort in the second pass.   ◇

This is a trivial example. There's no general recipe for how to implement the forward planner strategy—it's an art to work out how much working to keep, and how to keep it without creating extra bookkeeping work for ourselves.

---

[19]The binomial heap is also a 'forward planner' algorithm, it's just not gone all the way with forward planning for decreasekey.