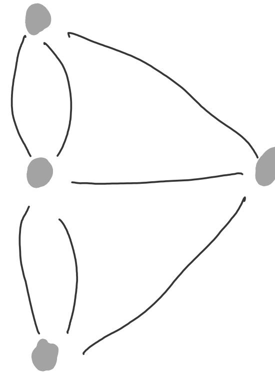
5. Graphs

5.1. Notation and representation

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

Example

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



Example

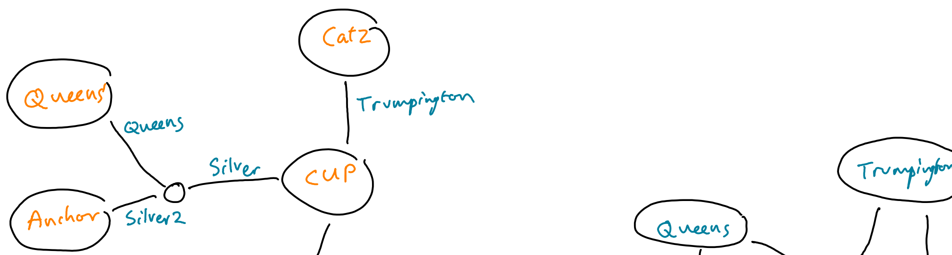
OpenStreetMap represents its map as XML, with nodes and ways.

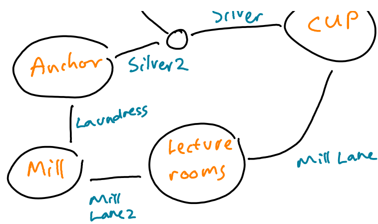
```
<osm version="0.6" generator="Overpass API">
  <node id="687022827" lat="52.2082725" lon="0.1379459" user="François Guerraz"/>
  <node id="687022823" lat="52.2080972" lon="0.1377715" user="bigalxyz123"/>
  <node id="687022775" lat="52.2080032" lon="0.1376761" user="bigalxyz123">
    <tag k="direction" 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>
```

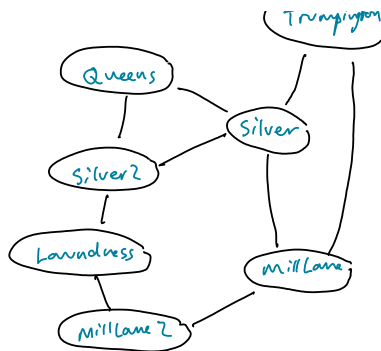
In some parts of the city, this data is very fine-grained. The more vertices and edges there are, the more space it takes to store the data, and the slower the algorithms run. Later in this course we will discuss geometric algorithms which could be used to simplify the graph while keeping its basic shape.

Example

It's up to you to decide what counts as a vertex and what counts as an edge.







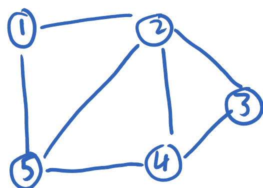
Definitions

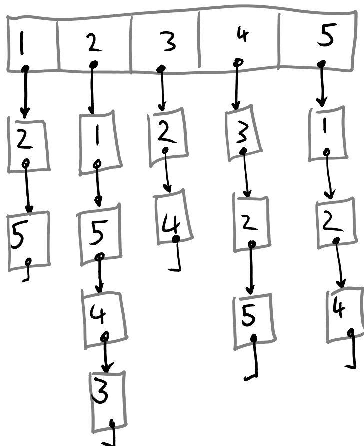
There are many different types of graph. Here are some common definitions.

- The graph is referred to as $G=(V,E)$ where V is the set of vertices and E the set of edges. Write $(v_1,v_2)\in E$ to refer to an edge.
- Sometimes the vertices or the edges have associated weights or other labels, as in the OpenStreetMap example.
- The graph may be **directed** in which case (v_1,v_2) means an edge from v_1 to v_2, also written $v_1\to v_2$. Or it may be **undirected** in which case (v_1,v_2) is treated the same as (v_2,v_1).
- We will not allow multiple edges between the same pair of vertices, unless stated otherwise. We will allow edges from a vertex back to itself, unless stated otherwise.
- A **path** is a sequence of vertices connected by edges. A path from a vertex back to itself is a **cycle**. A graph is **connected** if for every pair of vertices there is a path between them.
- Graphs without cycles have special importance, and **DAG** stands for Directed Acyclic Graph. In a DAG, if $v_1\to v_2$ we say that v_1 is the **parent** of v_2, or equivalently v_2 is the **child** of v_1.
- An undirected graph that has no cycles and is connected is called a **tree**. If it is not connected it is a **forest**.

Representation

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





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

5.2. Depth first search

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

- A web crawler for a search engine: a vertex is a page, and an edge is a hyperlink. Follow all the links you can, and retrieve every page, and add it to your search index.
- Path finding. To find a path from one vertex v_0 to some other vertex v_1: start at v_0 and traverse the graph, remembering your path, stopping when you reach v_1.
- Component finding: assign each disconnected component of a graph a different colour.

General idea

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

With a couple of tweaks, this gives us an algorithm for traversing a graph. Keep exploring new edges whenever you find them. Mark the vertices you've visited, so if you come across them again you don't retread your steps. Keep track of which vertices you' ve seen but not completely explored, so you can come back to them.

Implementation

We could keep track of the vertices we're waiting to explore using a stack.

```
1     def dfs(g, s):
2         # Visit all the vertices reachable from s
3         for v in g.vertices:
4             v.seen = False
5         toexplore = Stack([s])
6         s.seen = True
7
8         while not toexplore.is_empty():
9             v = toexplore.popright()
10            print "visiting", v
11            for w in v.neighbours:
12                if w.seen: continue
13                toexplore.pushright(w)
14                w.seen = True
```

dfs_stack

Or we could implement it using recursion, i.e. use the language's call stack rather than our own.

```
1     def dfs(g, s):
2         # Visit all the vertices reachable from s
3         for v in g.vertices:
4             v.visited = False
5         visit(s)
6
7     def visit(v):
8         v.visited = True
9         print "visiting", v
10        for w in v.neighbours:
11            if w.visited: continue
12            visit(w)
```

dfs_recurse

We can tweak the dfs_recurse code to find a path between a pair of nodes. All it takes is a few extra lines (labelled +) to k eep track of the path so far.

```
1     def dfs(g, s, t):
2         # Find a path from s to t, assuming one exists
3         for v in g.vertices:
4             v.visited = False
5 +       s.come_from = None
6         visit(s)
7 +       path = [t]  # a list with one element
8 +       while path[0].come_from is not None:
9 +           path.prepend(path[0].come_from)
10 +      return path
11
```

```
12      def visit(v):
13          v.visited = True
14          for w in v.neighbours:
15              if w.visited: continue
16 +            w.come_from = v
17              visit(w)
```

dfs_recurse_path

If the graph is disconnected, and we have access to the full list of vertices, we can repeatedly restart the search to ensure we visit all vertices.

```
1       def dfs(g):
2           # visit all vertices
3           for v in g.vertices:
4               v.visited = False
5           for v in g.vertices:
6               if v.visited: continue
7               print "starting dfs from", v
8               visit(v)
```

dfs_recurse_all

Analysis

In dfs_recurse, line 4 is executed for every vertex, lines 8--9 are executed at most once per vertex, and line 11 is executed for every edge out of each vertex that is visited. Thus the total running time is $\Theta(V+E)$.

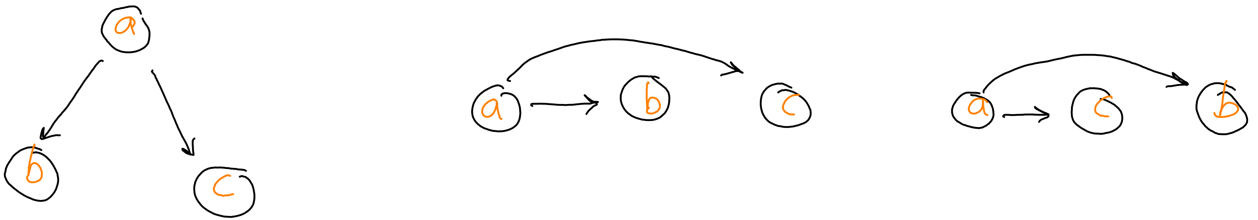
5.3. Topological sort

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

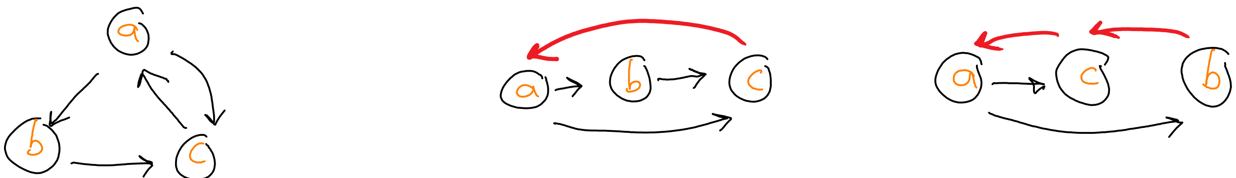
- Represent movies by vertices. Write $v_1 \to v_2$ to mean "The user has said she prefers v_1 to v_2". Is there a way to turn these pairwise preferences into a complete ranking, so that whenever $v_1 \to v_2$ then $\text{rank}(v_1)<\text{rank}(v_2)$? (Remember, rank=1 means top-ranked.)
- Deep learning systems like TensorFlow involve writing out the learning task as a collection of computational steps, each of which depends on the answers of some of the preceding steps. Write $v_1 \to v_2$ to mean "Step v_2 depends on the output of v_1." In what order should the steps be run?

Examples

It's easy to see that the total order might not be unique:



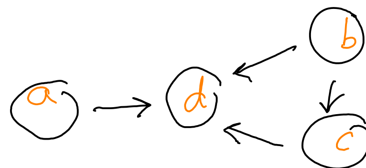
Does there exist a total order? If the graph has cycles, then no.



General idea

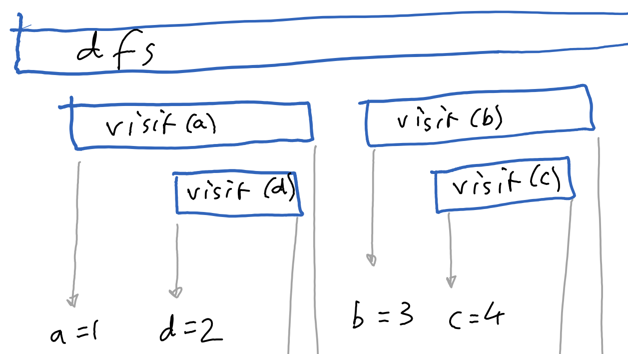
Depth-first search reaches a vertex v, then visits all its children and everything reachable from them. We want v to appear earlier in the ordering than its children and descendants. This suggests two possibilities: either (a) we assign the rank as soon as we visit a vertex (just after line 7 in dfs_recurse) and we increment the rank counter; or (b) we assign the rank when we leave a vertex (just after line 12) and we decrement the rank counter. With a tree, both possibilities work fine. With a general DAG, it's easy to find examples where (a) doesn't work.

an input graph

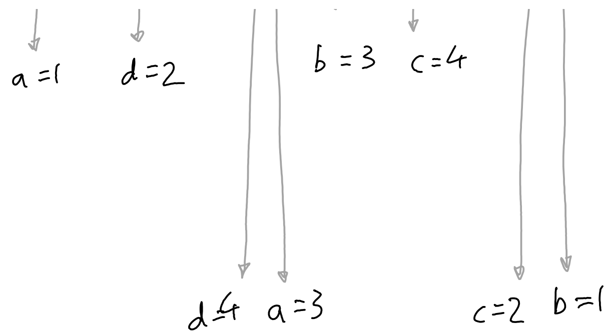


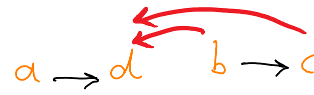
a possible call history, when running dfs

if we rank by

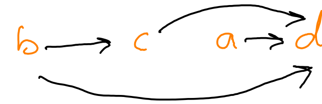


if we rank by when visit(·) starts





if we rank in reverse order of when visit(·) finishes



Problem statement

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

Implementation

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. There are also 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.

```
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 v.visited: continue
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 w.visited: continue
16               visit(w, totalorder)
17 +         totalorder.prepend(v)
18           # v.colour = 'black'
```

toposort

Analysis

We haven't changed the running time from that of dfs_recurse_all, which is $\Theta(V+E)$.

Theorem: The toposort algorithm terminates and returns totalorder which solves the problem statement.

Proof: Take two vertices v1 and v2 with an edge v1-v2. Imagine setting a breakpoint at line 13, to be triggered when v_1 becomes coloured grey. At this point, there are three possibilities for the colour of v_2:

1. If v_2 is black, then it must already have been prepended to `totalorder` in line 17, and v_1 hasn't yet been prepended, therefore v_1 will wind up earlier in `totalorder` than v_2.
2. If v_2 is white, i.e. not yet visited, then it will be visited in line 16 as one of v_1's neighbours. During this visit, it will be prepended to `totalorder`. The visit to v_2 will complete before we go on to line 17 and prepend v_1. Therefore v_1 will end up earlier in `totalorder` than v_2.
3. Suppose v_2 is grey. In this case, the call `visit`(v_1) must be nested inside the call `visit`(v_2), and the recursion stack gives us a path from v_2 to v_1. But we've assumed that $v_1 \to v_2$. Therefore there is a cycle. But we've assumed that g is a DAG, i.e. it has no cycles, which is a contradiction. Therefore this case cannot occur.

QED.

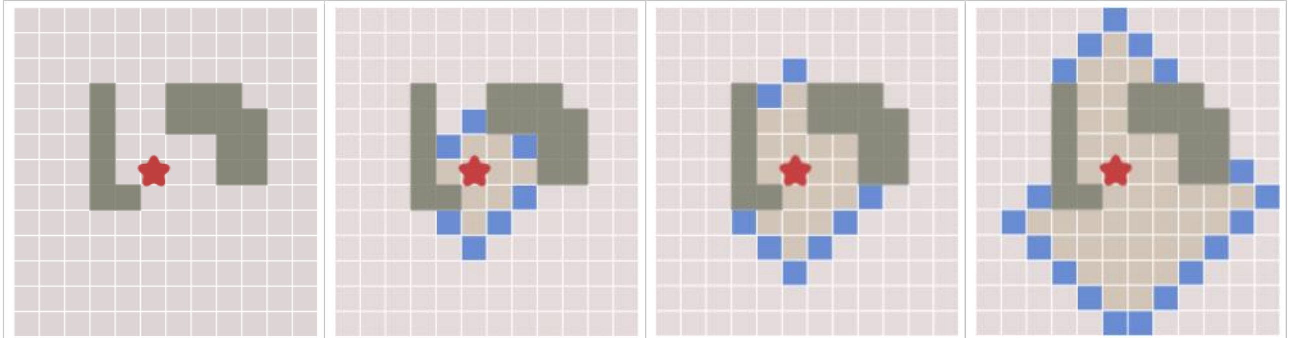
5.4. Breadth first search

Now we'll look at path-finding algorithms, of ever greater sophistication. We've seen that depth-first search can be used to find paths, though they may be very indirect. What about shortest paths?

General idea

Keep track of an expanding ring, called the *frontier*. Initially it contains just our start vertex. Repeatedly expand the frontier: first expand it to all vertices that are 1 hop from the start, then all vertices that are 2 hops, and so on. As soon as the frontier hits our end node, we've found a shortest path.

Here, the graph has a vertex for every light grey grid cell and edges between adjacent grid cells, and we're starting from the red blob.



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

Implementation

This code is almost identical to dfs_stack, the only difference being that we use a queue rather than a stack (changed lines labelled *). By using a queue, we will first visit all nodes in order of how many hops they are from the start vertex, and we don't actually need to keep track of the number of hops.

```
1       def bfs(g, s):
2           # Visit all the vertices reachable from s
3           for v in g.vertices:
4               v.seen = False
5 *         toexplore = Queue([s])
6           s.seen = True
7
8           while not toexplore.is_empty():
9               v = toexplore.popright()
10              print "visiting", v
11              for w in v.neighbours:
12                  if w.seen: continue
13 *                toexplore.pushleft(w)
14                  w.seen = True
```

bfs

5.5. 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 travel time: how do we find the quickest route between two locations?

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

Illustration

These pictures show two possible paths between the red blob and the purple cross. The left hand picture shows the number of hops from the red blob; the right picture shows the distance from the red blob. Here, dark grey cells can't be crossed, light grey cells cost 1 to cross, and green cells cost 5.

number of hops

5	4	5	6	7	8	9	10	11	12
4	3	4	5	6	7	8	9	10	11
3	2	3	4	5	6	7	8	9	10
2	1	2	3	4	5	6	7	8	9
1		1	2	3	4	5	6	7	8
2	1	2	3	4	5	6	7		9
3	2	3	4		6	7	8	9	10
4					7	8	9	10	11
5	6		10	9	8	9	10	11	12
6	7		11	10	9	10	11	12	13

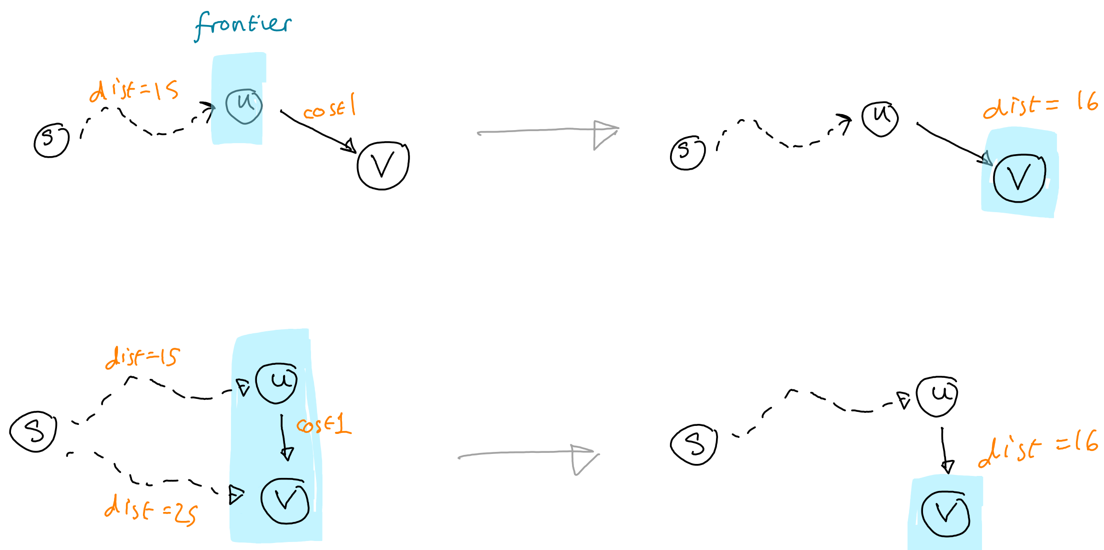
distance

5	4	5	6	7	8	9	10	11	12
4	3	4	5	10	13	10	11	12	13
3	2	3	4	9	14	15	12	13	14
2	1	2	3	8	13	18	17	14	15
1		1	6	11	16	21	20	15	16
2	1	2	7	12	17	22	21		17
3	2	3	4		22	23	18	17	18
4					27	24	19	18	19
5	6		30	29	26	21	20	19	20
6	7		25	24	23	22	21	20	21

General idea

In breadth-first search, we visited vertices in order of how many hops they are from the start vertex. Now, we'll 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 need to examine), but now we'll order it by distance rather than by number of hops. We'll use a *priority queue* to store the vertices in the frontier.

We might end up coming across a vertex multiple times, with different costs, so we need to change the logic. We'll add the vertex to the frontier either if we've never come across it, or if we've come across it already and our new path is shorter than the old path.



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.

Implementation

This algorithm was invented in 1959 and is due to Dijkstra (1930--2002), an influential pioneer of computer science, and an idiosyncratic character famous for his way with words. Some of his sayings:

- *The question of whether Machines Can Think [...] is about as relevant as the question of whether Submarines Can Swim.*
- *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.*

Line 5 declares that `toexplore` is a priority queue where the priority of an item v is `v.distance`. Line 10 iterates through all the vertices w that are neighbours of v, and retrieves the cost of the edge v→w at the same time.

```
1   def dijkstra(g, s):
2       for v in g.vertices:
3           v.distance = infinity
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: continue
14              w.distance = dist_w
15              if w in toexplore:
16                  toexplore.decreasekey(w)
17              else:
18                  toexplore.push(w)
```

dijkstra

Although we've called the variable *v*.distance, we really mean "shortest distance from *s* to *v* that we've found so far". It starts at ∞ and it decreases as we find new and shorter paths to *v*. It's easy to see that the algorithm never sets *v*.distance to be *less* than the true minimum distance from *s* to *v*, since it only ever looks at legitimate paths from *s*.

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.

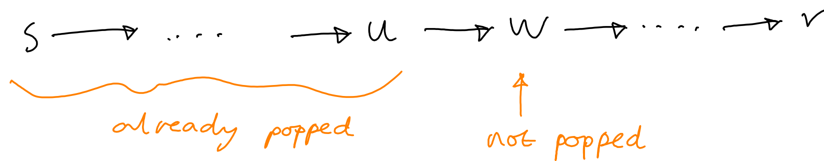
Analysis

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

Theorem. When run on a graph as described in the problem statement, the algorithm terminates. When it terminates then, for every vertex *v*, *v*.distance is equal to the distance from *s* to *v*.

Proof. Assertion on line 10 guarantees termination. It follows from the assertion on line 9 — once we've popped v then the only way v could be pushed back on line 18 is if we found a shorter path to v, which is impossible if the assertion is correct. So, all we need is to prove the assertion on line 9.

Proof of assertion line 9. Suppose it fails at some point in the execution. Let v be the vertex for which it first fails. Consider a shortest path from s to v,



The path starts at s, then there may be zero or more vertices that have already been popped, then there is at least one vertex that hasn't yet been popped, as in the picture (It may be that $w=v$.)

So,

$\text{distance}(s \text{ to } v) < v.\text{distance}$ — the alg. only sets v.distance when it has a possible path from s, so $\text{distance}(s \text{ to } v) \leq v.\text{distance}$. And the assertion failed at v, so the inequality is strict.

$\leq w.\text{distance}$ — w was put into toexplore by u, and it hasn't yet been popped, and the priority queue gave us v rather than w

$\leq u.\text{distance} + \text{cost}(u \to w)$ — when u was popped, and we looked at u's neighbours, we got this value for w.distance

$= \text{distance}(s \text{ to } u) + \text{cost}(u \to w)$ — since u did not fail the assertion when it was popped

$\leq \text{distance}(s \text{ to } v)$ — since $s \rightsquigarrow u \to w$ is on a shortest path from s to v,

and all edges have cost ≥ 0.

This is a contradiction. Thus Assertion on line 9 never fails.

$\square$

5.6. Bellman-Ford

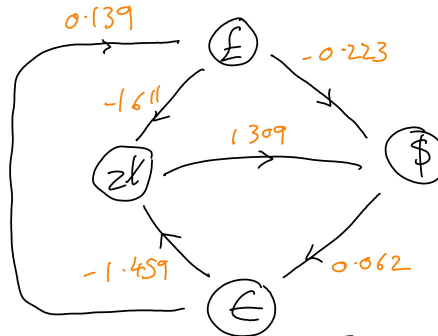
In some applications, we have graphs where some edge weights are negative. This is useful where we think of vertices as states that an entity can be in, and edges as actions that take it from one state to another, and where some actions have costs and others have rewards.

(The word 'distance' suggests a positive number, so we'll switch to the more neutral word *weight*.)

Examples

Let vertices represent currencies. If you can exchange 1 unit of v_1 for x units of v_2, put an edge $v_1 \to v_2$ with weight $-\log(x)$, which may be positive or negative. We'd expect that every cycle from a vertex back to itself would have positive weight, i.e. that sequence of currency exchanges loses money. If however we find a cycle of negative weight, then there is an arbitrage opportunity.

£1 → $1.25
$1 → €0.94
€1 → 4.30 zł
1 zł → $0.27
€1 → £0.87
£1 → 5.01 zł

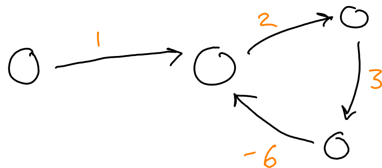


The cycle zł → $ → € → zł has weight -0.088

1 zł → $0.27 → €0.25 → 1.09 zł

Example: cycles with negative weight

For this graph, Dijkstra's algorithm does not terminate: it visits the three vertices in the cycle over and over again, reducing their distance scores each time, and each time putting the next vertex in the cycle back into `toexplore`.



General idea

Same as for Dijkstra: for any edge u→v,

s ⇢ u → v

minweight from s to v ≤ minweight from s to u + weight(u→v)

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

Problem statement

Given a directed graph where each edge is labelled with a weight, and a start vertex *s*, (i) if the graph contains no negative-weight cycles reachable from *s* then for every pair of vertices compute the weight of the minimal-weight path between those vertices; (ii) otherwise detect that there is a negative weight cycle reachable from *s*.

Implementation

In this code, lines 7 and 11 iterate over all edges in the graph, and `c` is the weight of the edge `u`→`v`. The assertion in line 9

refers to the true minimum weight among all paths from *s* to *v*, which it doesn't know yet; the assertion is just there to help us reason about how the algorithm works, and it's not part of the execution.

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

bellman_ford

Analysis

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

Theorem. The algorithm correctly solves the problem statement. In case (i) it terminates successfully, and in case (ii) it throws an error in line 13.

Proof. Write $w(v)$ for the true minimum weight among all paths from s to v, with $w(v) = -\infty$ if there is a path that includes a -ve -weight cycle. The algorithm only ever looks at valid paths to v, therefore the assertion on line 9 is true.

Proof for case (i). Pick any vertex v, and consider a minimal-weight path from s to v. This path can't have cycles (if it did, the cycle has weight ≥ 0 by assumption, so we could cut it out). So the path has at most $V-1$ edges. (Remember, V = #vertices) Let it be

$$s = v_0 \to v_1 \to \cdots \to v_k = v.$$

During the first pass of lines 7-8, we'll look at edge $v_0 \to v_1$, so we'll achieve

$$v_1.\text{minweight} = v_0 + \text{weight}(v_0 \to v_1)$$
$$= w(v_1) \quad (v_0 \to v_1 \text{ must be an optimal path to } v_1)$$

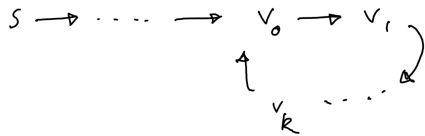
By the second pass, we'll achieve

$$v_2.\text{minweight} = w(v_2)$$

and so on. By the time it reaches line 11, it's found $v.\text{minweight} = w(v)$ for every vertex v. The test in line 12 must fail, because

for every vertex v. The test in line 12 must fail, because otherwise we could construct a path from s to v of weight less than v.minweight. Therefore the algorithm terminates successfully.

Proof for case (ii) Suppose there is a -ve-weight cycle



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

If the algorithm did not throw an error then all these edges passed the test in line 12, ie

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

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

$$\vdots$$

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

Putting all these equations together,

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

thus the cycle has weight ≥ 0 This contradicts the premise — hence at least one of the edges failed the test in line 12.

□

5.7. Johnson's algorithm

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

- Each router in the internet has to know, for every packet it might receive, where that packet should be forwarded to. Path preferences in the Internet are based on link costs set by internet service providers. Routers send messages to each other advertising which destinations they can reach and at what cost. The Border Gateway Protocol (BGP) specifies how they do this. It is a distributed path-finding algorithm, and it is a much bigger challenge than computing paths on a single machine.

- The *betweenness centrality* of an edge is defined to be the number of shortest paths that use that edge, from all the shortest paths between all pairs of vertices in a graph. (If there are n shortest paths between a pair of vertices, count each of them as contributing $1/n$.) The betweenness centrality is a measure of how important that edge is, and it's used for summarizing the shape of e.g. a social network. To compute it, we need shortest paths between all pairs of vertices.

General idea

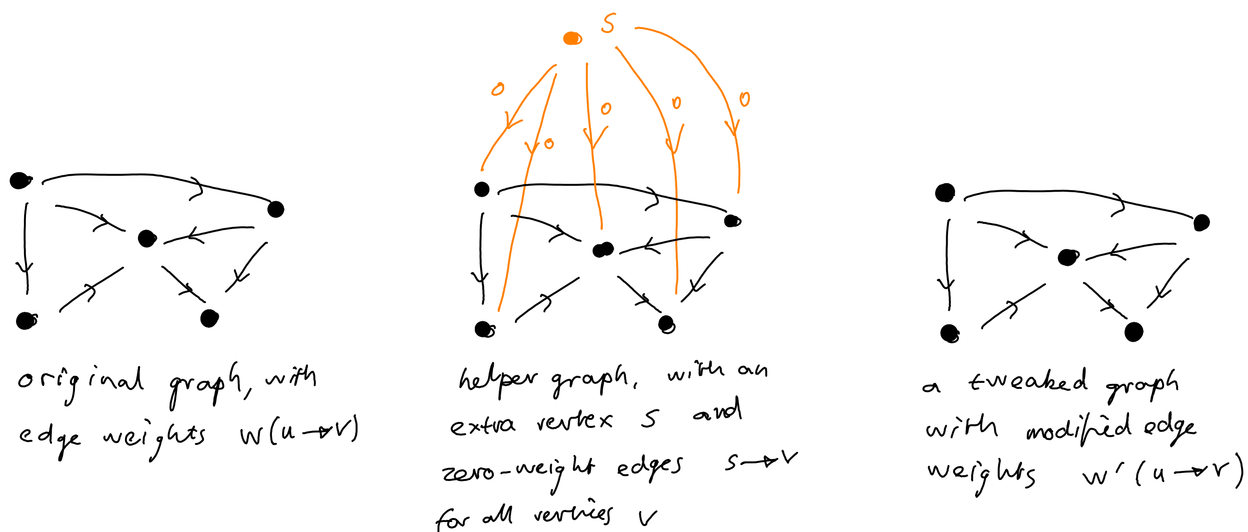
If all edge weights are ≥ 0, we can just run Dijkstra's algorithm V times, once from each vertex. The running time is $O(V^2 \log(V) + V E)$, which is just V times the running time of Dijkstra.

If some edge weights are < 0, we could run the Bellman-Ford algorithm once from each vertex, and the total running time would be $O(V^2 E)$. But there is a clever trick, published by Donald Johnson in 1977, whereby we can run Bellman-Ford once, then run Dijkstra once from each vertex, then run some cleanup for every pair of vertices. The running time is therefore $O(V^2 \log(V) + V E) + O(V E) + O(V^2)$ which is $O(V^2 \log(V) + V E)$.

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. Build a helper graph, as illustrated.
   Run Bellman-Ford to find minimum weights of paths from s to every other vertex v, call it d_v.

   (obviously $d_v \leq 0$, using the direct link $s \to v$. But if there are negative

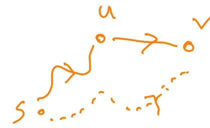
weights, some vertices will have $d_v < 0$.)

If Bellman-Ford finds a -ve-weight cycle from s, then the original graph has a -ve-weight cycle; report this and exit. If not, then the original graph has no -ve-weight cycle, so proceed.

2. Define a tweaked graph by setting

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

CHECK. Is $w' \geq 0$?



we know $d_v \leq d_u + w(u \to v)$

therefore $w'(u \to v) = d_u + w(u \to v) - d_v \geq 0$.

CHECK. Does the tweaked graph have the same min-weight paths as the original?

Pick any two vertices p and q.

For any path $p = v_0 \to v_1 \to \cdots \to v_k = q$ between them,

$$\text{weight in tweaked graph} = \left(d_{v_0} + w(v_0 \to v_1) - d_{v_1}\right) + \left(d_{v_1} + w(v_1 \to v_2) - d_{v_2}\right) + \cdots$$

$$= d_p + w(v_0 \to v_1) + w(v_1 \to v_2) + \cdots + w(v_{k-1} \to v_k) - d_q$$

$$= \text{weight in original graph} + \underbrace{d_p - d_q}$$

a constant that doesn't depend on the path

Thus the tweaked graph has the same ranking of paths as the original (though the weights are different).

3. To recover minimum weights for the original graph,

$$\text{min weight from p to q.} = \text{min weight from p to q} - d_p + d_q$$

from p to q in original graph = from p to q in tweaked graph $- a_p + a_q$

5.8. All-pairs shortest paths with matrices

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

General idea

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

Let $M^{(n)}$ be a VxV matrix, where $M^{(n)}_{ij}$ = minimum weight among all paths from i to j that have n or fewer edges.

We can write out an equation for $M^{(n)}$ in terms of $M^{(n-1)}$, and this leads directly to an algorithm for computing $M^{(n)}$. We still need to figure out how large n has to be, and that turns out to be easy.

Problem statement

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

Analysis

Define the $V \times V$ matrix W by

$$W_{ij} = \begin{cases} 0 & \text{if } i=j \\ \infty & \text{if there is no edge } i \to j \\ \text{weight of the edge } i \to j & \text{otherwise} \end{cases}$$

Then $M^{(1)} = W$, and

$$M^{(n)}_{ij} = M^{(n-1)}_{ij} \wedge \left[(M^{(n-1)}_{i1} + W_{1j}) \wedge (M^{(n-1)}_{i2} + W_{2j}) \wedge \cdots \wedge (M^{(n-1)}_{iV} + W_{Vj}) \right]$$

$x \wedge y$ means $\min(x,y)$

the best path of length $n-1$

go from i to 2 in $\leq n-1$ hops, then from 2 to j

$$= (M^{(n-1)}_{i1} + W_{1j}) \wedge \cdots \wedge (M^{(n-1)}_{iV} + W_{Vj})$$

since $M^{(n-1)}_{ij} = M^{(n-1)}_{ij} + W_{jj}$

This is just like matrix multiplication,

$$M^{(n)} = M^{(n-1)} \otimes W$$

except we're doing $+$ instead of $\cdot$ and $\wedge$ instead of $+$.

As with matrix multiplication, it takes V^3 operations to do $\otimes$.

As with Bellman-Ford,

(1) compute $M^{(V-1)}$ and $M^{(V)}$

(2) If they are different, the graph has a negative-weight cycle

(3) If they are equal, we've found the minimum weights.

There's a cunning trick. To illustrate, suppose $V = 10$. Rather than 8 multiplications to compute $M^{(9)}$, we can repeatedly square:

$$M^{(1)} = W$$
$$M^{(2)} = M^{(1)} \otimes M^{(1)}$$
$$M^{(4)} = M^{(2)} \otimes M^{(2)}$$
$$M^{(8)} = M^{(4)} \otimes M^{(4)}$$
$$M^{(16)} = M^{(8)} \otimes M^{(8)} = M^{(9)}$$, if there are no -ve-weight cycles.

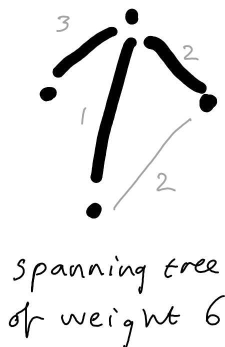
This trick gives an overall running time

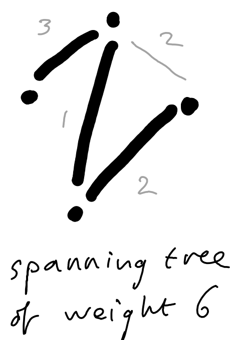
$$O(V^3 \log V)$$

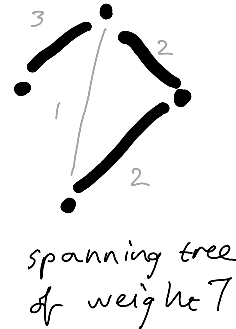
5.9. Prim's algorithm

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

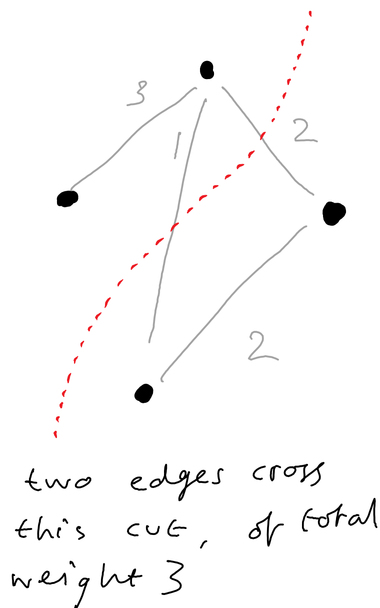
- For clustering: let vertices represent the things we want to cluster, and let the weight of an edge measure the dissimilarity between its vertices. If we delete the *k*-1 edges of largest weight, we're left with *k* clusters. This can be used, for example, to find regions in an image.
- The MST problem 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?

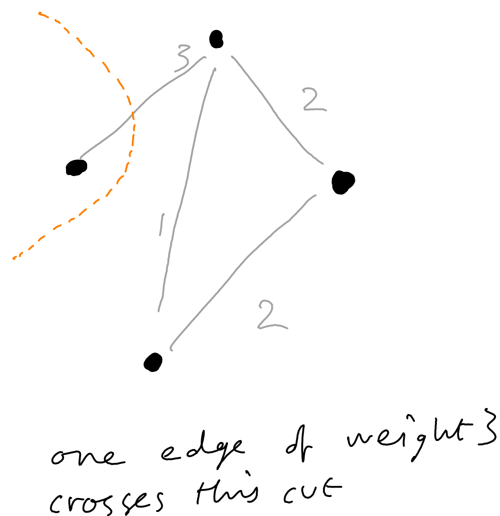






For solving the MST problem, and for other problems to do with constructing networks on top of graphs, it's useful to make a definition: a **cut** of a graph is an assignment of its vertices into two non-empty sets, and an edge **crosses** the cut if its two ends are in different sets.

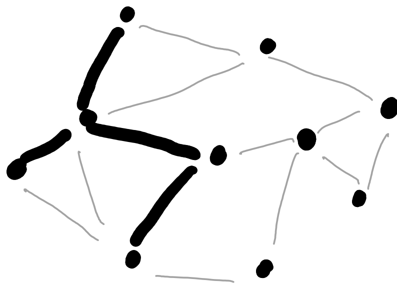


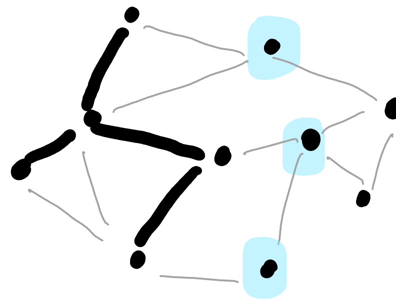


General idea

We'll build up the MST greedily. Pick an arbitrary vertex to start the tree. Every iteration, we have some vertices which are already part of the tree, and a *frontier* of vertices which aren't in the tree yet but which are neighbours of some vertex in the tree. Pick the edge from the tree to the frontier that has lowest weight, and add it to the tree. Repeat until there are no vertices left. (Recall Dijkstra's algorithm, which used a frontier in a very similar way.)

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





Problem statement

Given an undirected graph *g* with edge weights, construct an MST.

Implementation

This code is very similar to Dijkstra's algorithm. There are some extra lines to keep track of the tree (labelled +), and two modified lines (labelled *) because here we're interested in 'distance from the tree' whereas Dijkstra is interested in 'distance from the start node'. The start vertex *s* can be chosen arbitrarily. The algorithm is due to Jarnik (1930), and independently to Prim (1957) and Dijkstra (1959).

Line 7 declares that `toexplore` is a priority queue where the priority of an item `v` is `v.distance`. For the vertices in the frontier, this records the distance from that vertex to the tree we've built so far. Line 15 iterates through all the vertices `w` that are neighbours of `v`, and retrieves the weight of the edge `v—w` at the same time.

```
1       def prim(g, s):
2           for v in g.vertices:
3               v.distance = infinity
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, and edges
13              #     {w--w.come_from   for w in g.vertices}
14              # Assert: t is part of an MST
15              for (w, edgeweight) in v.neighbours:
16 *                if w.in_tree or edgeweight > w.distance: continue
17 *                w.distance = edgeweight
18 +                w.come_from = v
19                  if w in toexplore:
20                      toexplore.decreasekey(w)
21                  else:
22                      toexplore.push(w)
```

prim

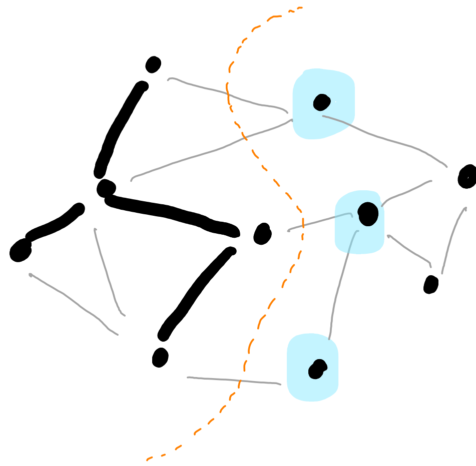
Analysis

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

Theorem. Whatever the start vertex *s*, Prim's algorithm terminates, and the set of edges {*v*—*v*.come_from for all vertices *v* except *s*} forms an MST.

At line 11, the algorithm adds a new vertex to the tree, via the minimum weight edge from the tree to the rest of the graph. Think of this as a cut: on one side the vertices already in the tree, on the other side the vertices not in the tree, and we're picking the minimum weight edge across the cut. The following lemma says that the enlarged tree is still part of an MST, i.e. the assertion on line 14 is true. Thus, when it terminates, the algorithm has produced a full MST.

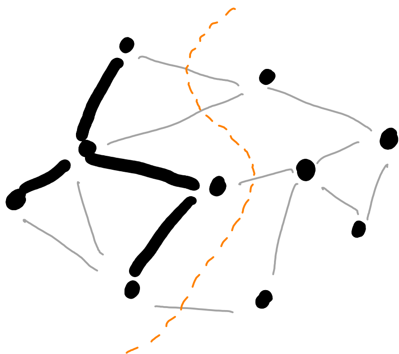
minimum weight edge across the cut. The following lemma says that the enlarged tree is still part of an MST, i.e. the asserti on on line 14 is true. Thus, when it terminates, the algorithm has produced a full MST.

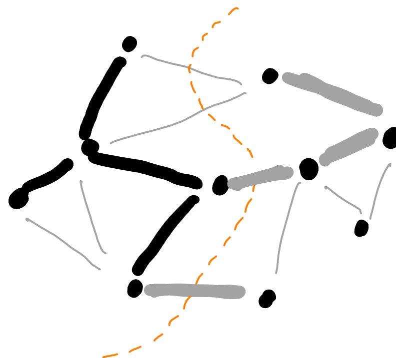


the cut between the tree we've built so far and the rest of the vertices

Lemma. Let *g* be a connected graph with edge weights, let *f* be a subset of the edges of an MST, and consider a cut of the vertices of *g* such that no edges in *f* cross the cut. Let *e* be the minimum weight edge of *g* that crosses the cut. Then, *f* ∪ {*e*} is also a subset of the edges of an MST.

Proof



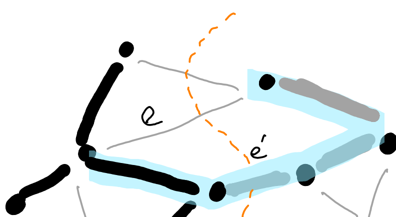


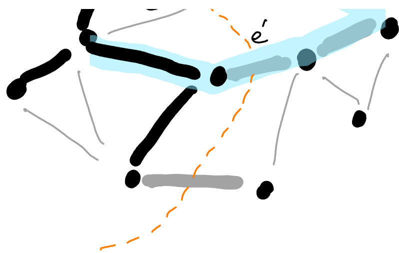
f (in black), and an MST that f is part of (grey).

Write $\bar{f}$ for the MST that f is part of.

If $\bar{f}$ contains e, we are done.

If not, let e = v—u, and consider a path in $\bar{f}$ connecting v to u. This path doesn't contain e (since e is not in $\bar{f}$), so it must use some other edge e′ across the cut





Now consider the graph f', with the edges of $\bar{f}$, but with e added and e' removed. Since e is the minimum-weight edge across the cut, weight$(f') \leq$ weight(f). It is not hard to check that f' is also a spanning tree.

Therefore, f' is an MST that contains $f \cup \{e\}$, as required.

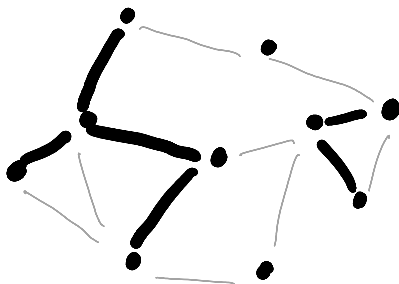
$\square$

5.10. Kruskal's algorithm

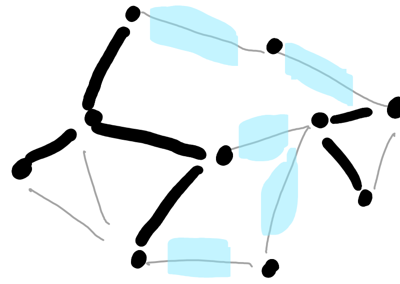
Another algorithm for finding a minimum spanning tree is due to Kruskal (1956). It makes the same assumptions as Prim's algorithm, and it has the same running time.

General idea

Kruskal's algorithm builds up the MST by agglomerating smaller subtrees together. At each stage, we've built up some fragments of the MST, and the algorithm joins the two fragments that have the lowest-weight edge between them.



four fragments have been found so far



five candidate edges to use, to join together two of the fragments

Implementation

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

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

```
1   def kruskal(g):
2       tree_edges = []
3       partition = DisjointSet()
4       for v in g.vertices:
5           partition.addsingleton(v)
6       edges = sorted(g.edges, sortkey = lambda u,v,edgeweight: edgeweight)
7
8       for (u,v,edgeweight) in edges:
9           p = partition.getsetwith(u)
10          q = partition.getsetwith(v)
11          if p == q: continue
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: the connected components of f correspond to vertex-sets in partition.
16          # Assert: f is part of a MST
17
18      return tree_edges
```

kruskal

Analysis

The running time of Kruskal's algorithm depends on how DisjointSet is implemented. We will see (later in the course) that the operations on DisjointSet can all be done in $O(1)$ time, giving total cost $O(E \log E)$ for the sort on line 6, and $O(V + 3E) = O(E)$ for everything else, so the total running time is $O(E \log E)$. Since the maximum possible number of edges in an undirected graph is $V(V-1)/2$, $\log E = O(\log V)$, and so the running time can equivalently be written $O(E \log V)$.

To prove that Kruskal's algorithm finds an MST, we apply the lemma used for the proof of Prim's algorithm, as follows. When the algorithm merges *p* and *q*, consider the cut of g's vertices into *p* versus not-*p*; the algorithm picks the lowest weight edge across

this cut, and so by the lemma we've still got an MST.