D2. Weighted Graphs

Notes by Malcolm Brooks,
partly inspired by notes of Pierre Portal.

Text Reference (Epp)	3ed: Chapter 11
	4ed: Chapter 10
	5ed: Chapter 10

Some of the work in this section is not covered in our text by Epp.
I have based some examples on ones from:
Kolman, Busby & Ross *Discrete Mathematical Structures*
Johnsonbaugh *Discrete Mathematics*

Weighted graphs

- In some applications of graphs there is a non-negative number associated with each edge, known as the **weight** of the edge.
- So a **weighted graph** is a graph G together with a **weight function** weight : $E(G) \to \mathbb{Q}_+$.
- Examples include:
  - **Airline networks:** Vertices are airports used; edges are services or potential services between airports; weights are air miles or cost of flights.
  - **Fibre telecommunications:** Vertices are primary routing stations; there are edges between every pair of vertices (a complete graph); weights are the costs of laying fibre between stations.
  - **The internet:** Vertices are internet nodes; edges are all direct connections between nodes; weights are times (in milliseconds) for a packet to travel across a connection.

Problems on weighted graphs

We define the **(total) weight** of a subgraph *S* of a weighted graph *G* to be the sum of the weights of all its edges.

There are several well-studied optimisation problems relating to subgraphs of weighted graphs. We will look at just three:

Mininimal spanning tree:

Find a spanning tree of least possible total weight.

Travelling salesman problem:

Find a Hamilton circuit of least possible total weight.

Shortest path:

Find a path between two given vertices that has least possible total weight.

We will also look at a different kind of problem on a weighted **directed** graph: Maximal Flow. Details later.

Kruskal's algorithm for minimal spanning tree

Kruskal's algorithm is an obvious modification to the spanning-tree-finding algorithm we discussed previously. We simply choose the 'cheapest' possible edge at each step:

Input: Weighted connected graph G with n vertices.

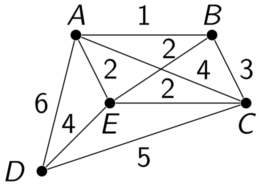
Output: Minimal spanning tree T for G.
Total weight W of this tree.

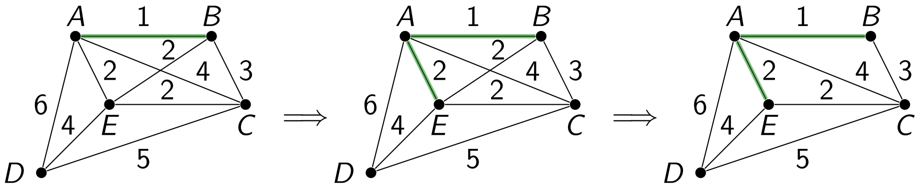
Method:

1. Initialise T to have all the vertices of G but no edges. Initialise W to 0.
2. From the edges of currently in G pick one, e, of least weight and remove it from G.
3. If adding e to T does not create a circuit in T, add e to T and add weight(e) to W.
4. Repeat steps 2 and 3 until T has $n - 1$ edges.

Example: Applying Kruskal's algorithm

Find a minimal spanning tree for this weighted graph:

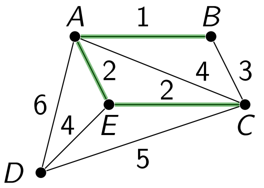




$W = 1$ $\qquad W = 3$ $\qquad W = 3$

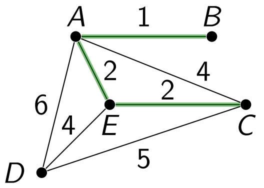
Example: Applying Kruskal's algorithm (cont.)

$\Longrightarrow$



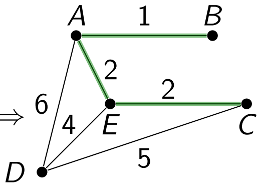
$W = 5$

$\Longrightarrow$



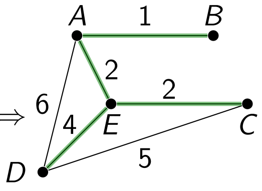
$W = 5$

$\Longrightarrow$



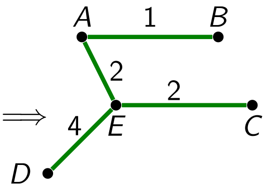
$W = 5$

$\Longrightarrow$



$W = 9$

$\Longrightarrow$



$W = 9$

(The generated spanning tree)

‘Greedy’ algorithms

- A ‘greedy’ algorithm is an algorithm for an optimisation problem that attempts to find a globally optimal solution by finding a locally optimal solution at each step.

- Kruskal’s algorithm, that we examined for solving the minimal spanning tree problem, is an example of a greedy algorithm because:

- Kruskal’s algorithm attempts to find a spanning tree of **least** possible *total* weight by, at each step, adding an edge of **least** possible (*individual*) weight (from amongst all unused edges that would not create a circuit).

- Kruskal’s algorithm always succeeds! (Non-obvious theorem omitted)

  That is, it always finds a minimal spanning tree, given any weighted connected (finite) graph.

The ‘Travelling salesman’ problem

- This problem originates from a time when all salespeople were men !!
- The salesman needs to visit n towns on a shortest possible ‘circular tour’.
- Given: a table of distances between every pair of towns.
- **Model:** Graph K_n with towns as vertices and edges weighted by the the inter-town distances.

  Find a Hamilton circuit of minimum possible total weight.

The ‘Nearest Neighbour’ algorithm (for the travelling salesman problem)

Input: Weighted complete graph G with n vertices.

Output: Hamilton circuit for G as a list L of vertices.
Total weight W of this circuit.

Method:
1. Initialise L to the empty list, W to 0 and index i to 1
2. Choose any vertex and append it to L as $L(1)$.
3. From all vertices in G but not in L, choose a vertex v such that weight$(L(i), v)$ is as small as possible. (v is a ‘nearest neighbour’ to $L(i)$).
4. Add weight$(L(i), v)$ to W. Increment i by 1. Append v to L as $L(i)$.
5. Repeat steps 3 and 4 until $i = n$.
6. Add weight$(L(n), L(1))$ to W. Append $L(1)$ to L as $L(n+1)$.

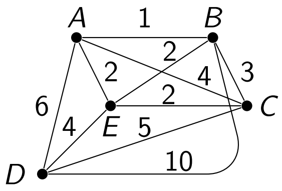
Greedy algorithms don’t always succeed

- The Nearest Neighbour algorithm is another example of a greedy algorithm because at each step it looks for the ‘best way out’, ignoring any possible disadvantages later on.
- But the Nearest Neighbour algorithm often fails to find the shortest Hamilton circuit.
- Greed doesn’t always pay !!
- In fact, no efficient successful algorithm for the travelling salesman problem is known at this time. Finding one, or proving that none exists, is a major outstanding problem in mathematics.

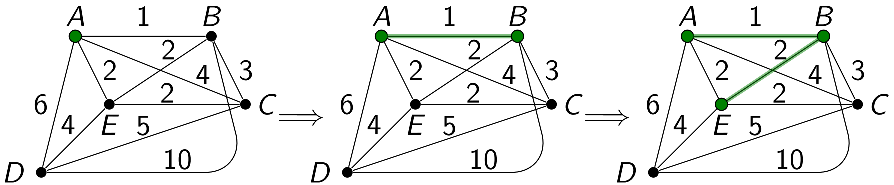
Example: Applying the Nearest Neighbour algorithm

Find a minimal Hamilton circuit (tour) for this weighted graph:

Note: This graph is as for the minimal spanning tree example but with the addition of an edge BD to make it complete.



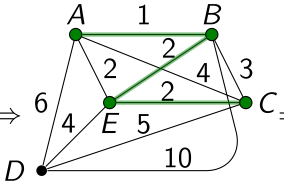
Let's start at A (we can start anywhere):



$L(1) = A,\ W = 0$ $\quad$ $L(2) = B,\ W = 1$ $\quad$ $L(3) = E,\ W = 3$

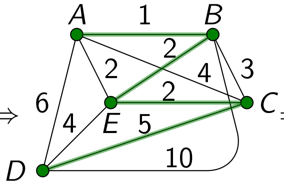
Example: Applying the Nearest Neighbour algorithm (cont.)

$\Longrightarrow$



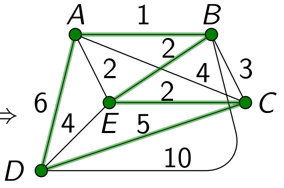
$L(4) = C, \; W = 5$

$\Longrightarrow$



$L(5) = D, \; W = 10$

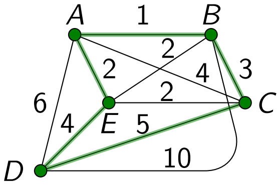
$\Longrightarrow$



$L(6) = A, \; W = 16$

So Nearest Neighbour has generated a tour of weight 16.
However this is NOT minimal.
The weight of a minimal tour is in fact 15.

Here is a tour of minimal weight:

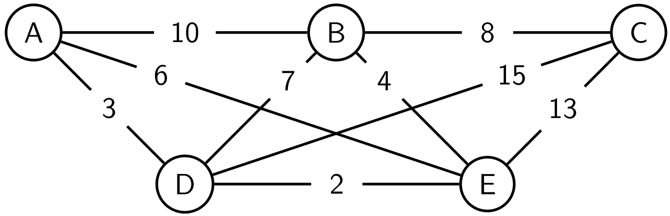


$W = 15$

Note that Nearest Neighbour may generate this tour if we start at D instead of A. Then $L(2) = E$ and it just depends on the choice for $L(3)$.

Shortest Path — Introduction

Consider this weighted graph:



Problem: Find a shortest path from A to C .

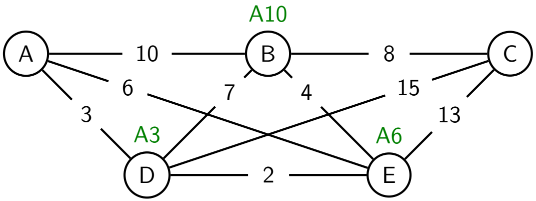
That is, from the many paths from from A to C find one whose total weight is as small as possible.

For a small graph like this you can soon find a shortest path just by trying many alternatives (there are 10 or so simple A→C paths here).

Can you spot a shortest path?

For large graphs this approach is not practical. We need an *algorithm*.

Dijkstra's Algorithm



Edsger Dijkstra 1930 - 2002

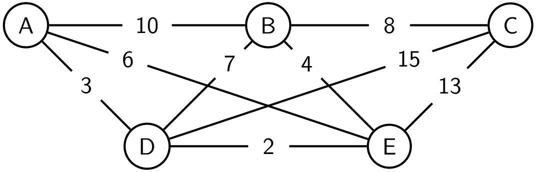
Did you notice the special layout of the graph?

As will be explained, when using Dijkstra's algorithm (for finding a shortest path) markers and labels are inserted above vertices (green text in the above example). The special layout helps to avoid these annotations getting mixed up with the main graph information.

Dijkstra's algorithm has some similarities to Kruskal's algorithm for finding a minimum spanning tree but is a little more complicated. So I will delay setting out the algorithm formally and launch straight in to demonstrating its use on the above example.

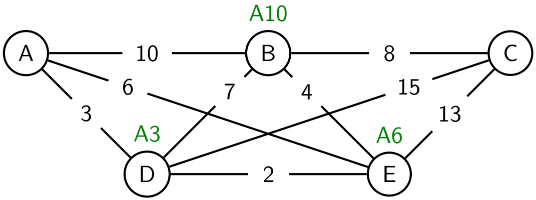
Example 1 — Slide 1

We seek a minimal weight path from A to C in the graph below.



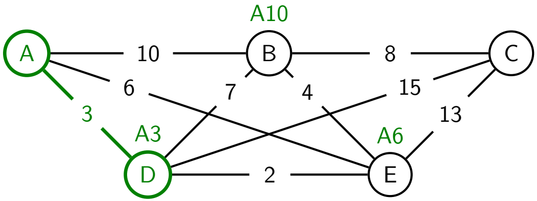
In particular this will yield the minimal 'distance', via graph edges, of C from A. In fact the algorithm proceeds by progressively finding the minimal distance of many, possibly all, vertices from A.

Start by labelling each vertex adjacent to A with its 'direct' distance from A:



Example 1 — Slide 2

Vertex D carries the smallest of these distances. 'Lock it in' and also the edge leading to it (I use heavy lines to indicate locking):



We call vertex D the current vertex c.

Next mark and label any un-locked vertex v adjacent to the current vertex c.

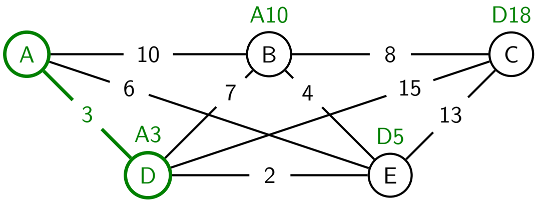
There are always three possibilities, and all three occur here:

Possibility 1: v is unmarked. *e.g.* Vertex C .
Mark v with c and label with its distance from A via c.
So we write D18 above C.

Possibility 2: *v* needs updating. *e.g.* Vertex E .
The distance to *v* via *c* is less than the distance currently shown. Remark with *c* and relabel with the shorter distance.
So we replace A6 above E with D5

Possibility 3: *v* needs no updating. *e.g.* Vertex B.
The distance to *v* via *c* is no less than the distance currently shown.
So we leave the A10 above B as it is.

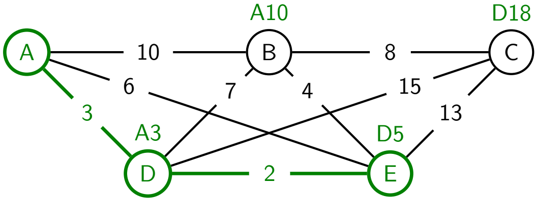
The annotated graph now looks like this:



Example 1 — Slide 4

We now have three so-called 'fringe' vertices, B, C and E.
Fringe vertices are those that have been annotated (marked and labelled) but not yet locked in.

Now locate and lock in a fringe vertex v with the lowest label value.
That's vertex E in our example since $5 < 10$ and $5 < 18$.
Also lock in its marked lead-in edge. That's edge DE for us.



The latest locked-in vertex E becomes the new current vertex.
We now repeat the process applied to the previous current vertex D.

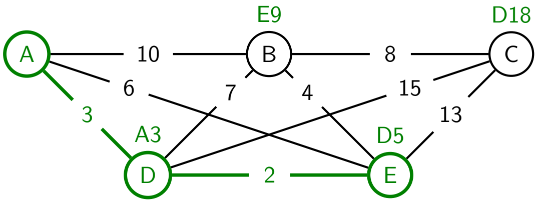
Example 1 — Slide 5

Vertex E is adjacent to two un-locked vertices, B and C:

Vertex B needs updating, since $5 + 4 = 9 < 10$. So it is re-marked with E and re-labelled 9; *i.e.* A10 is replaced by E9.

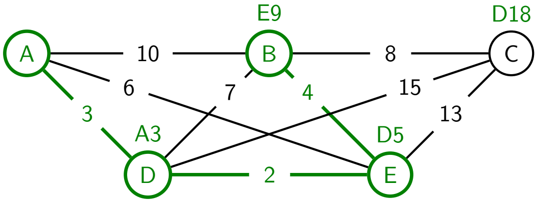
Vertex C does not need updating since $5 + 13 = 18$, and C is already labelled 18.

The updated graph now looks like this:



The lowest fringe value is now 9 on B, so we lock in B and its lead-in edge EB (next slide).

Example 1 — Slide 6



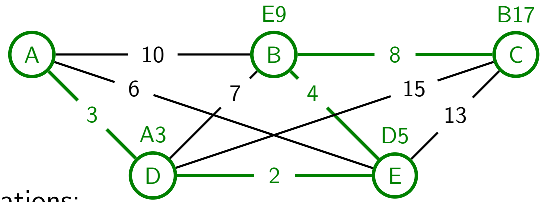
The new current vertex is the just locked-in B.

There is only one vertex adjacent to B that has not already been locked in, namely C. This needs updating since $9 + 8 = 17 < 18$.
So D18 is replaced by B17.

Since vertex C is the only vertex in the fringe it has the lowest label by default. So C and its lead-in edge BC are locked in.

We have now locked in the minimal distance 17 into our 'target' vertex C, so we can stop. The final diagram is .....

Example 1 — Slide 7; Results and Comments



Some Observations:

- Besides the shortest path from A to C, the solution provides the shortest path to all the vertices along that path. For this example that happens to be the entire vertex set.
- Had there remained un-locked vertices, we could have continued the process until there were none.
- Since no vertex is locked twice, the locked edges form a tree. The required shortest path is the unique path on that tree from A to C.
- With all vertices locked, the solution provides a spanning tree for the graph.

Dijkstra's Algorithm — A Formal Description

Input: (a) Connected simple graph G. Vertices A, Z from G.
(b) Distance function $\mathrm{dist}\colon E(G) \to \mathbb{Q}^+$.

Output: (a) Tree T containing A and Z as vertices.
T is a subgraph of G.
The unique path A→Z in T has minimal total distance of all paths A→Z in G.
(b) 'Labelling' $L\colon V(T) \to \mathbb{Q}_+$; $L(v) = \text{min.dist}(\mathrm{A} \to v)$.

Method:

```
1. Initialize the tree $T$:  Set $V(T) = \{\mathrm{A}\}$, $E(T) = \emptyset$.
2. Initialize a 'Marking' function $M\colon V(G) \to V(G) \cup \{\text{blank}\}$:
   Set $M(v) = \text{blank}$ for all $v \in G$.
3. Set $L(\mathrm{A}) = 0$. Set 'current vertex' $c$ to A.
While $c \neq \mathrm{Z}$:
    4. For each vertex $v$ adjacent to $c$ but not in $T$:
       If $v$ is unmarked (i.e. $M(v) = \text{blank}$)
       or if $L(v) > L(c) + \mathrm{dist}(\{c, v\})$
       set $M(v) = c$, $L(v) = L(c) + \mathrm{dist}(\{c, v\})$.
```

Dijkstra's Algorithm — A Formal Description (cont.)

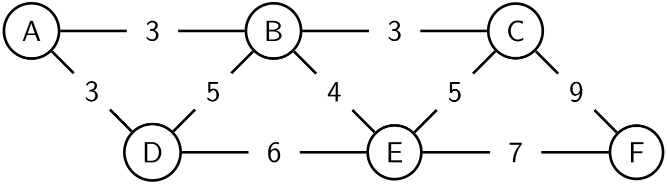
```
        5. From all marked $v \in G \setminus T$ (i.e. $M(v) \neq$ blank and $v \notin T$)
           (such $v$ are said to be ‘on the fringe’)
           select one, say $w$, with minimal $L(v)$.
        6. Insert vertex $w$ and edge $\{M(w), w\}$ into the tree $T$.
           (I call this “locking in” $w$ and its lead-in edge.)
        7. Update $c$ to $w$. (i.e. make $w$ the new current vertex.)
    End of While Loop
  End of Method
```

This completes the formal description of Dijkstra's shortest path algorithm.

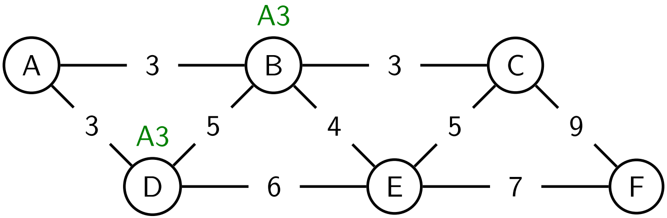
Next a second example, but this time with less commentary.

Example 2 – Slide 1

Find the shortest path for A to F:

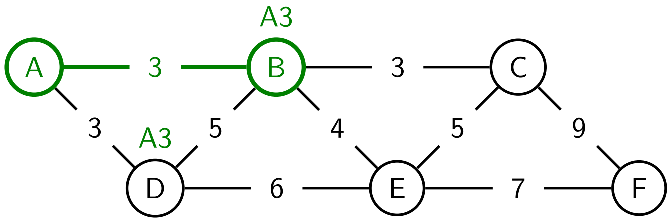


First annotate the vertices adjacent to the start vertex A:

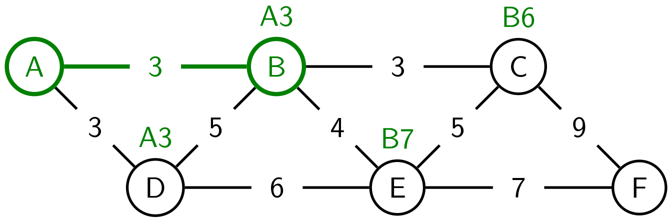


Vertices B and D have equal lowest label; let's lock in B:

Example 2 – Slide 2

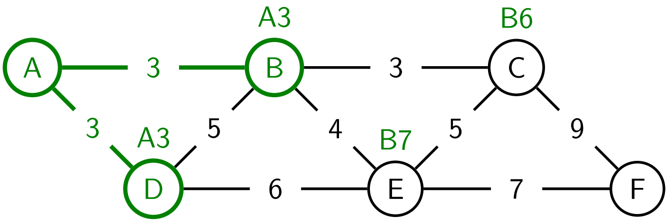


Current vertex is now B. Fringe vertices will be C,D,E.
Annotations required for C and E but D's does not need updating.

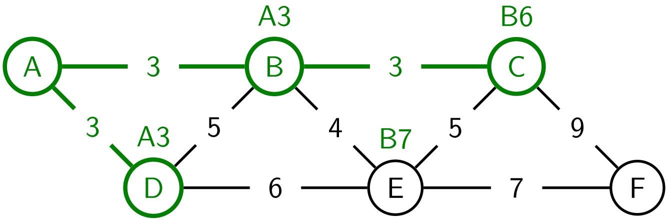


D now has lowest label so needs locking in next:

Example 2 – Slide 3

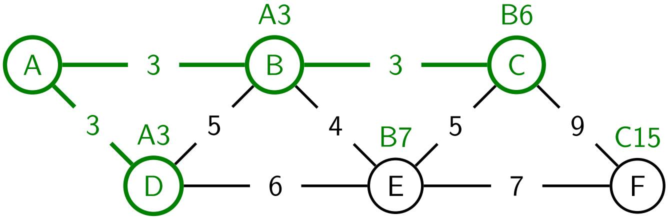


Current vertex is now D. Its only un-locked neighbour is E but no updating is required. The fringe vertices are C and E. Vertex C has lower label value so it is next to be locked:

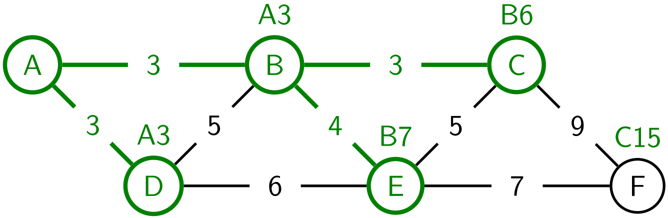


Example 2 – Slide 4

Of the two un-locked vertices adjacent to C, E is marked but does not need updating while F is unmarked and so needs annotating:

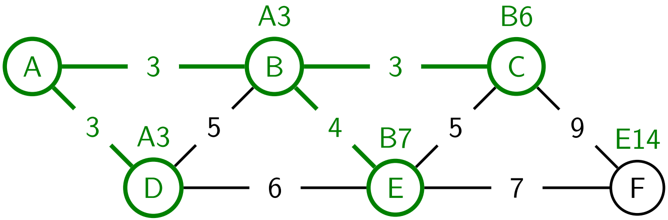


Of the two fringe vertices, E has the lower label value so is locked in. Its lead-in vertex is marked as B, so edge BE is also locked in.

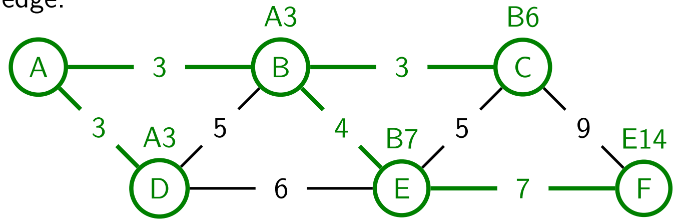


Example 2 – Slide 5

The new current vertex E has only one un-locked neighbour, F, and F needs updating since $7+7<15$:



Now F is the only fringe vertex, so F is locked in, together with its lead-in edge.

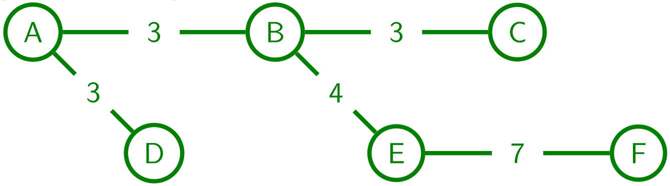


Example 2 — Slide 6; Results and Comment

Since the ‘target’ vertex F has now become the current vertex, the algorithm terminates. From the annotation E14 on F we read off that the the shortest ‘distance’ between A and F is 14.

Backtracking using the annotations gives $F \longleftarrow E \longleftarrow B \longleftarrow A$. That is, the shortest path from A to F is ABEF.

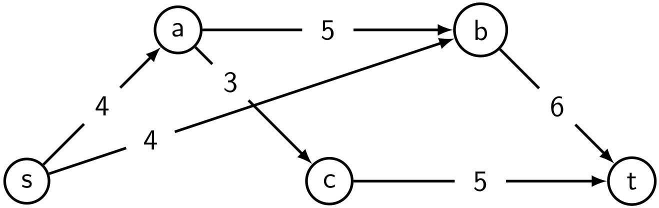
As can often happen with small examples, the algorithm has locked in all vertices of the graph, meaning that the locked-in edges provide a spanning tree for the graph:



As it happens, this is a minimal spanning tree. However, in general a spanning tree produced by Dijkstra’s algorithm will not be minimial.

Transport networks

The digraph below is an example of a simple **transport network** :



The defining features of a simple transport network are:

- The edges are weighted with positive weights, called **capacities**.
- There is exactly one special vertex called the **source** (labelled ‘s’) which is not the end vertex of any edge.
- There is exactly one special vertex called the **sink** or **target** (labelled ‘t’) which is not the start vertex of any edge.
- Every edge lies on some simple (directed) path from s to t.

Flows

The motivation for transport networks is to model the *flow* of something, such as gas or information, through a network of links.

A simple example to keep in mind is water flowing from a reservoir (the source) through various pipes (the edges) and exchange nodes (the vertices) to another reservoir (the sink, or target).

An 'active' transport network has a flow $F(e) \geq 0$ through each edge e. The **flow** F, like the capacities C, is a function $E(D) \to \mathbb{Q}_+$, where D is the digraph representing the network. However, each edge e has a *fixed* capacity $C(e)$ whereas its flow $F(e)$ can vary, subject to constraints:

(1) Flow cannot exceed capacity. $[\forall e \in E(D) \;\; F(e) \leq C(e).]$

(2) In each edge, flow direction = edge direction.

(3) Total flow into a node equals total flow out, except for nodes s, t. $[\forall v \in V(D) \backslash \{s, t\} \;\; \sum_{e \in v_{\text{in}}} F(e) = \sum_{e \in v_{\text{out}}} F(e)$, where $v_{\text{in}}, v_{\text{out}}$ are the sets of edges coming **in** to, and **out** of, v, respectively.]

Finding a maximum flow

For any flow F the constrains on the network imply that the total flow out of the source must equal the total flow into the target:

$$\sum_{e \in s_{\text{out}}} F(e) \;=\; \sum_{e \in t_{\text{in}}} F(e) \;=\; v(F) \text{ say.}$$

This common total $v(F)$ is the **flow value** and any flow achieving the maximum possible flow value is called a **maximum flow**, $\boldsymbol{F_{\text{max}}}$. (There may be several flows that achieve the maximum.)

I will demonstrate a vertex labelling algorithm for finding an F_{max}. The algorithm starts with the **zero flow** $\boldsymbol{F_0}$, where $\forall e \in D\ F_0(e) = 0$. Then, over a number of stages, expanded flows $F_1, F_2, \ldots$ are obtained until F_{max} is reached.

At stage i, flow F_i is constructed as $F_i = F_{i-1} + f_i$, where the incremental flow f_i is based on a constant $k_i \in \mathbb{Q}^+$ and a simple path p_i from s to t:

$$f_i(e) = \begin{cases} k_i & \text{for every edge } e \text{ on the path } p_i \\ 0 & \text{for evey other edge } e. \end{cases}$$

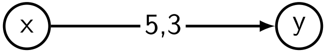
Spare capacity

In the actual lecture I skip this slide and the next, jumping straight to the first example and explaining notation and terminology as they arise. That makes them more digestible I believe.

Giving a formal description of the vertex labelling algorithm at this point may be a little overwhelming, so I will start with an example. But I need to first explain my terminology and mark-up conventions.

Given a transport network D with capacity and flow functions C, F, we know that $F(e) \leq C(e)$ for every edge $e \in E(D)$. I will call the non-negative difference $S(e) = C(e) - F(e)$ the **spare capacity** of the edge e. (Some authors use the term "excess capacity".)

To depict a flow I will follow the capacity value $C(e)$ on each (directed) edge e with the flow value $F(e)$ for that edge. For example



represents a flow of 3 in the edge from x to y, with spare capacity 2.

Levels and labels

At each stage of the vertex labelling algorithm levels and labels are associated afresh with the vertices of the network.

The **level** of a vertex is determined iteratively as follows:

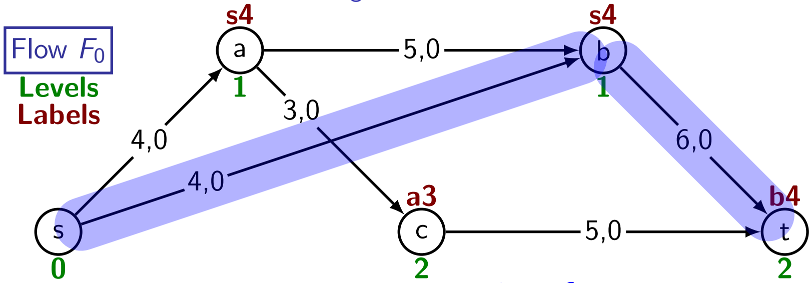
- The source vertex s always has level 0.
- If $e = (s, x)$ and $S(e) > 0$ then x has level 1.
- If x has level n and $S((x, y)) > 0$ then y has level $n + 1$ provided it has not already been assigned a lower level.

The **label** on a vertex v of level $n \geq 1$ has the form uk, where u is a vertex of level $n-1$ and $(u,v) \in E(D)$ is an edge on the path for a potential incremental flow through v with flow value k.

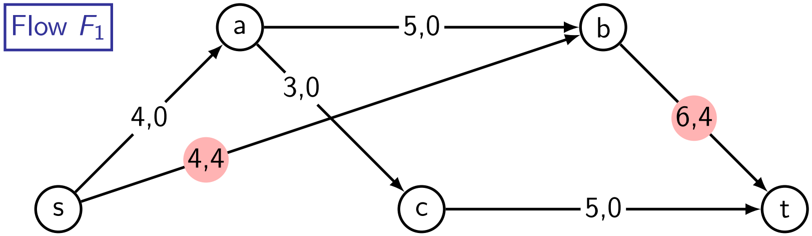
The algorithm assigns labels in ascending order of levels, and in alphabetical order within levels. Exactly how u and k are determined is explained in the formal description of the algorithm, but you may be able to infer the rule from the next example.

Vertex labelling algorithm, Example 1

Stage 1: F_0 to F_1

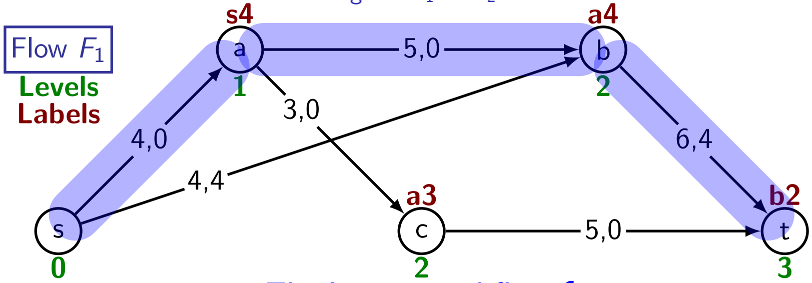


The incremental flow f_1

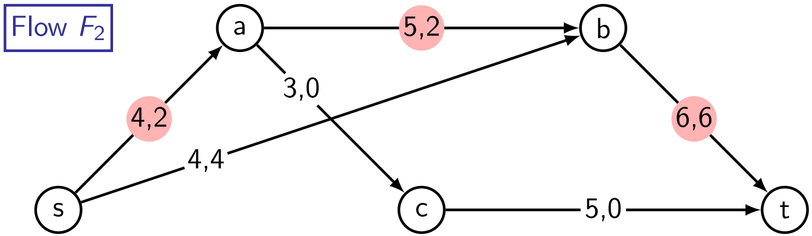


Vertex labelling algorithm, Example 1

Stage 2: F_1 to F_2



The incremental flow f_2



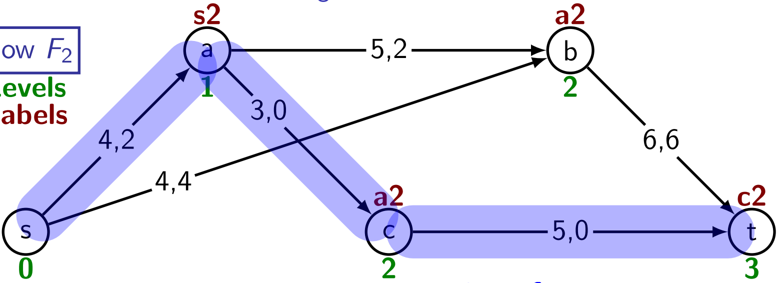
Vertex labelling algorithm, Example 1

Stage 3: F_2 to F_3

Flow F_2

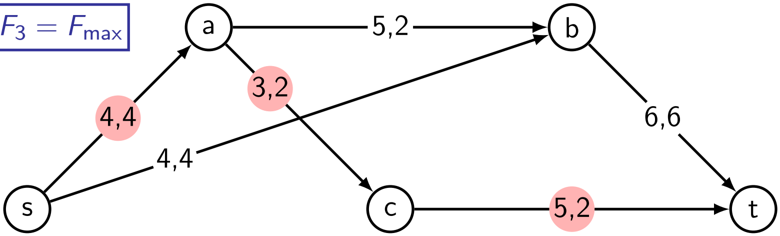
Levels

Labels



The incremental flow f_3

Flow $F_3 = F_{\max}$



Vertex labelling algorithm, Formal description

The algorithm is unavoidably complicated to describe fully.
Moreover a further complication (virtual capacity) still needs adding.

Input: Transport network D with capacity function C.

Output: A maximum flow function $F_{\max}$ for the network.

Method: Initialise F to the zero flow F_0. Initialize i to 1.
For $i = 1, 2, \ldots$ carry out stage i below to attempt to build an incremental flow f_i.
If stage i succeeds, define $F_i = F_{i-1} + f_i$ and proceed to stage $i+1$.
If stage i fails, define $F_{\max} = F_{i-1}$ and stop.

Stage i:

1. If $i > 1$, mark up the amended edge flows for F_{i-1}.
2. Mark up the levels for F_{i-1}, as explained earlier.
3. Next slide....

3. If t is assigned a level, stage i will succeed, so continue.
   If not, then stage i fails, so return above to define F_{max} and terminate.
4. Mark up labels for F_{i-1} as follows until t is labelled:
   (a) Label each level 1 vertex v with sk_{v}, where $k_{\mathrm{v}} = S((s,v))$.
   (b) If t has level 2 or more now work through the level 2 vertices in alphabetical order, labelling each vertex v with uk_u, where
      - u is the alphabetically earliest level 1 vertex with $(u,v) \in E(D)$ and $S((u,v)) > 0$,
      - k_{v} is the minimum of $S((u,v))$ and the value part of u's label.

   (c) If t has level 3 or more now work through the level 3 vertices in a similar manner and so on.
5. Let p_i be the path $\mathrm{u}_0\mathrm{u}_1\ldots\mathrm{u}_n$ where $\mathrm{u}_n =$ t and for $0 < j \leq n$ u_j has label $\mathrm{u}_{j-1}k_j$.
   Define f_i to be the incremental flow on p_i with flow value k_n.

End of Method

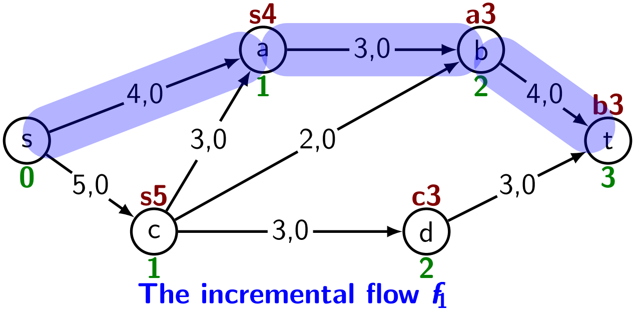
Vertex labelling algorithm, Example 2

Stage 1: F_0 to F_1

Flow F_0

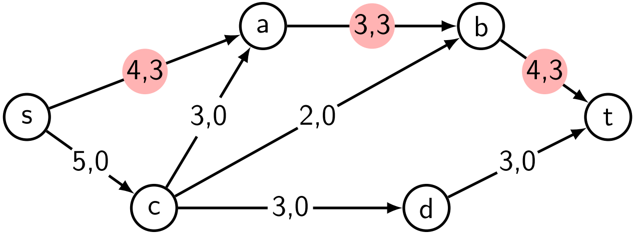
Levels

Labels



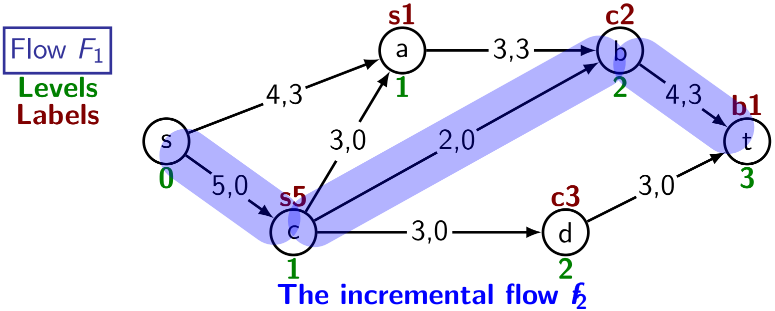
The incremental flow f_1

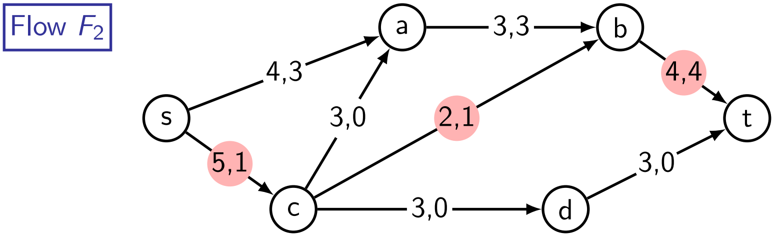
Flow F_1



Vertex labelling algorithm, Example 2

Stage 2: F_1 to F_2





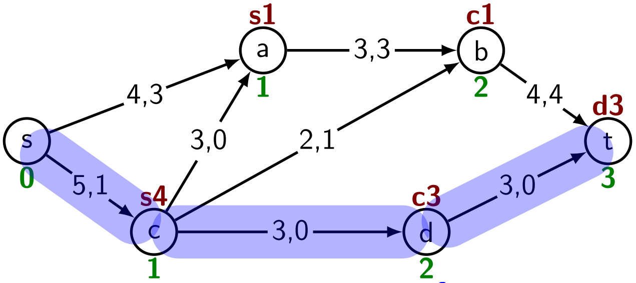
Vertex labelling algorithm, Example 2

Stage 3: F_2 to F_3

Flow F_2

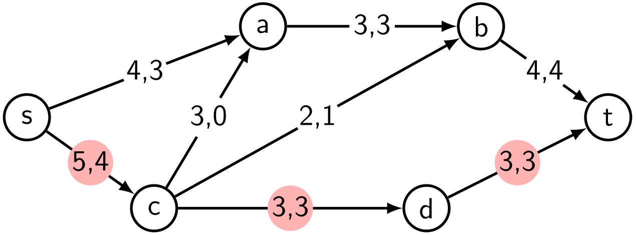
Levels

Labels



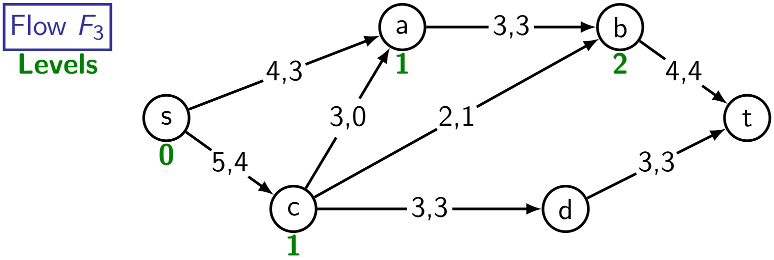
The incremental flow f_3

Flow F_3



Vertex labelling algorithm, Example 2

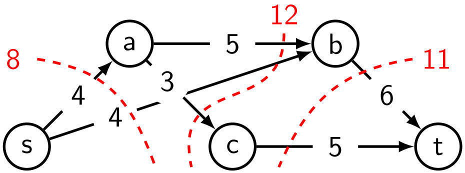
Stage 4: F_3 is $F_{\max}$



No level can be assigned to t !

So the algorithm terminates with $F_{\max} = F_3$.

Cuts



Consider again Example 1 at right.
Since the total capacity of edges leading from the source is $4 + 4 = 8$ it is clear that the maximum flow value cannot be more than 8.

In fact the maximum flow value *is* 8 as a flow with that value is easily found. (We found it using the vertex labelling algorithm, but that isn't really needed on such a simple example.)

The set of edges leading out of the source is an example of a network *cut*, so called because removing the edges would 'cut' the network in two, separating the source from the target.
The $4 + 4$ is the 'capacity' of the cut.

A cut may be indicated by drawing a (red) line through its edges.
Some examples, with values, are shown on the digraph above.

Max Flow Min Cut

Observe that for Example 1 the minimum cut and maximum flow have the same value (8). As you probably expect, this is in fact true for all transport networks, as given by the theorem below.

First a formal definition of 'cut':
A set K of edges in a transport network D is called a **cut** when there is a partition $\{S, T\}$ of $V(D)$ with $s \in S$ and $t \in T$ such that K comprises all edges of D that start in S and finish in T; *i.e.* $K = E(D) \cap (S \times T)$.

The **capacity of a cut** K is $\sum_{e \in K} C(e)$.

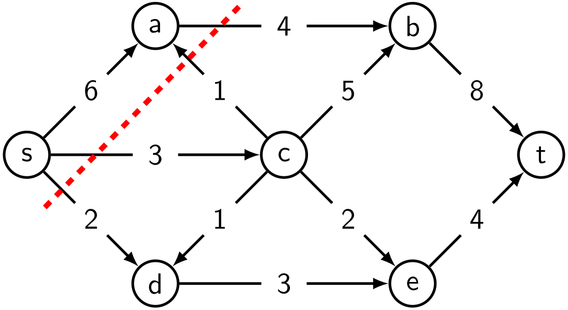
Max flow min cut theorem: For any transport network:

$$\boxed{\textbf{maximum flow value} \quad = \quad \textbf{minimum cut capacity}}$$

Though highly plausible, this theorem is little tricky to prove, and the proof will be omitted, as will the proof that the vertex labelling algorithm always finds a maximum flow.

Max flow min cut: Class example 1

What is the maximum flow value for this transport network?

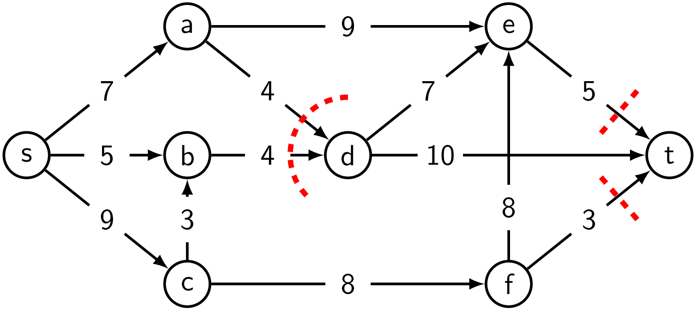


Answer: After some searching we find the minimum cut shown, for which $S = \{s, a\}$ and $T = \{b, c, d, e, t\}$.
The capacity of this cut is 4+3+2=9, so this is the maximum flow.
Note: Edge (c,a) is not in the cut since it's in the wrong direction.

Max flow min cut: Class example 2

from Kolman, Busby & Ross

What is the maximum flow value for this transport network?

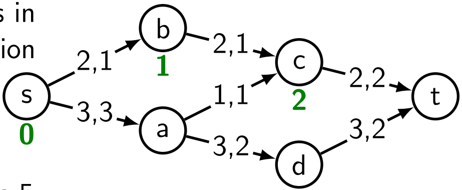


Hint: This time the minimum cut cannot be drawn as a single line!
Answer: Use all the edges from $S = \{s, a, b, c, e, f\}$ to $T = \{d, t\}$.
The capacity of this cut is $4+4+5+3=16$, so this is the max flow.

Introduction to virtual flows

The next example (Example 3) will show that in order to always achieve a maximum flow the labelling algorithm needs a modification.

Part way through assigning levels in Stage 4 (of Example 3) the situation becomes as shown at right. We are 'stuck' at vertex c, but so far we only have a flow value of 4, whereas the min cut value is 5.



To escape from this we need to 'divert' the flow a→c→t to a→d→t, thereby allowing another 1 unit of flow s→b→c→t.

The algorithm accomplishes this diversion by allowing a 'virtual flow' of 1 unit c→a so that, in particular, vertex a becomes level 3.
What happens after that is shown as Stage 4 unfolds in the full description of the use of the algorithm in Example 3 that follows.

First, the definition and an explanation of how the algorithm is modified.

The complete vertex labelling algorithm

Let (u,v) be a (directed) edge in a transport network D, and suppose there is currently a flow of $f > 0$ along this edge. The vertex labelling algorithm can reduce this flow to $g < f$ by introducing a **virtual flow** of $f - g$ in the opposite direction, *i.e.* from v to u.

Virtual flows are *always* in a direction counter to that indicated on the digraph D, and if part of an incremental flow will *reduce* the flow in the relevant leg of the path.

The modification to the algorithm, affecting the assignment of levels and labels, is an amendment to the definition of spare capacity.

For vertices u,v of D, where D has capacity and flow functions C, F:

$$\boxed{S((\text{u,v})) = \begin{cases} C((\text{u,v})) - F((\text{u,v})) & \text{if } (\text{u,v}) \in E(D) \\ F((\text{v,u})) & \text{if } (\text{v,u}) \in E(D) \\ 0 & \text{otherwise} \end{cases}}$$

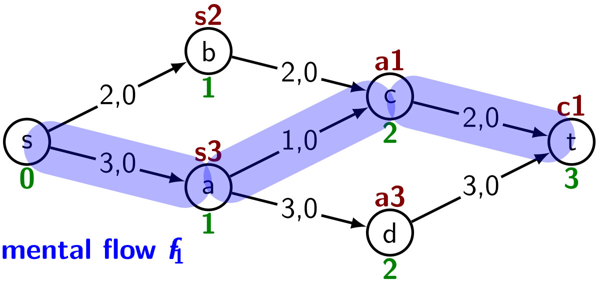
When $(\text{v,u}) \in E(D)$, $S((\text{u,v}))$ is called a **virtual capacity**.

Vertex labelling algorithm, Example 3

Stage 1: F_0 to F_1

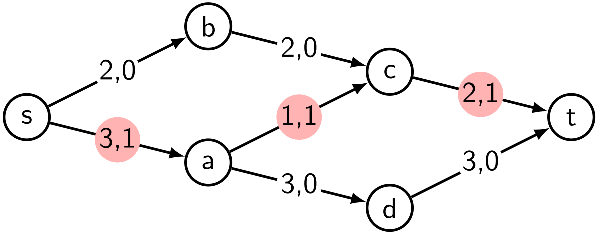
Flow F_0

Levels
Labels



The incremental flow f_1

Flow F_1

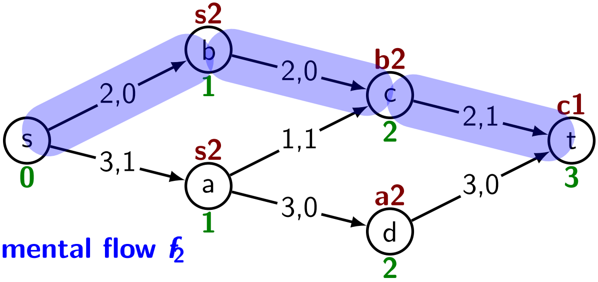


Vertex labelling algorithm, Example 3

Stage 2: F_1 to F_2

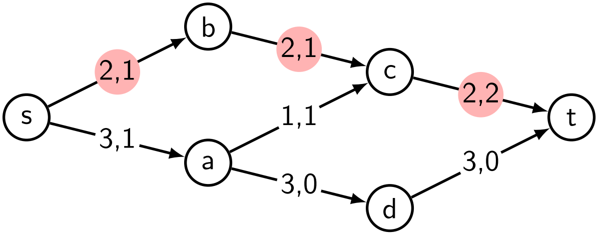
Flow F_1

Levels
Labels



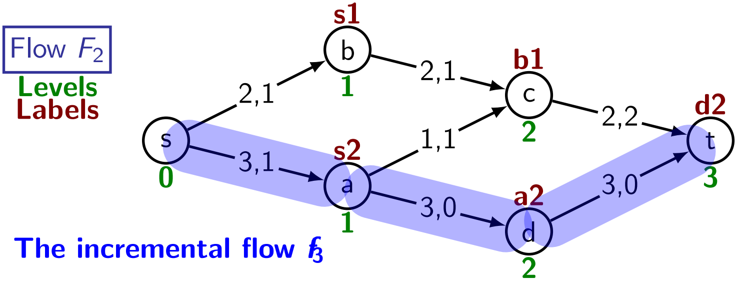
The incremental flow f_2

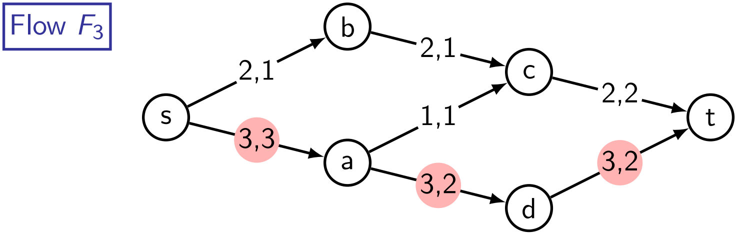
Flow F_2



Vertex labelling algorithm, Example 3

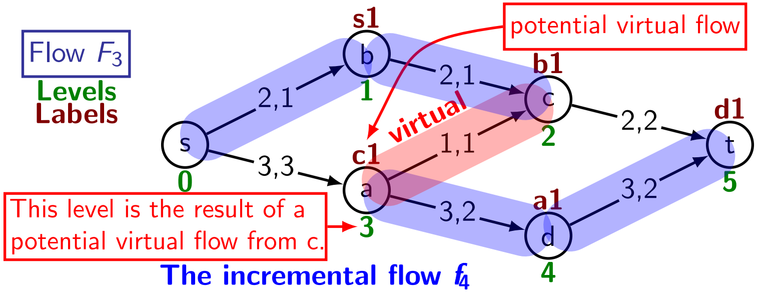
Stage 3: F_2 to F_3

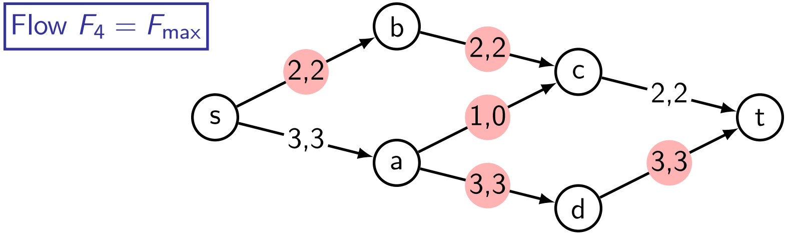




Vertex labelling algorithm, Example 3

Stage 4: F_3 to F_4





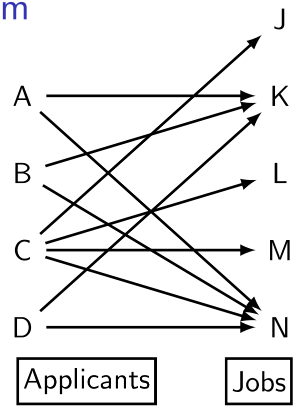
A matching problem

from Johnsonbaugh

Four people A,B,C,D are each interested in one or more of five jobs J,K,L,M,N on offer. The diagram indicates who is interested in what job. Is it possible to satisfy everyone?

That is, can each applicant x be *matched* with a job $m(x)$? If so, how?

If not, what's the best that can be done?



One answer:

x	A	B	C	D
$m(x)$	K	N	J	-

More generally, given a relation $R \subseteq S \times T$ a **matching problem** seeks a **maximal matching function** (or just a 'matching') $m \subseteq R$. This is a **injective** (one-to-one) function $f : S' \to T$ with domain $S' \subseteq S$ as large as possible subject to m being an injective subset of R.

Solving matching problems

The little 4-applicants-5-jobs matching problem can be easily solved 'by eye'. Larger matching problems require a more systematic method. Perhaps surprisingly, the vertex labelling algorithm for transport networks can be used.

The matching problem for $R \subseteq S \times T$ can be converted to the max flow problem for a transport network as follows:

1. Create a digraph D by adding to the arrow diagram for R
   - a **supersource** s and edges (s,x) for each $x \in S$ and
   - a **supertarget** t and edges (y,t) for each $y \in T$.
2. Assign capacity 1 to every edge of D. (*i.e.* $\forall e \in E(D)\ C(e) = 1$.)

A solution to the max flow problem provides the matching:

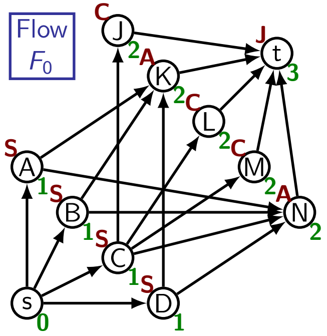
$$\boxed{m = \{(x, y) \in S \times T \ : \ F_{\max}((x, y)) = 1\}.}$$

Vertex labelling for matching; Example 1

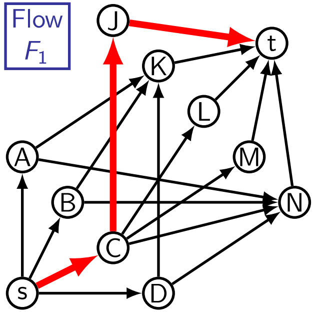
Here is how vertex labelling would be used on the Johnsonbaugh problem. This is just for demonstration, since, as we've seen, it's easily solved by eye.

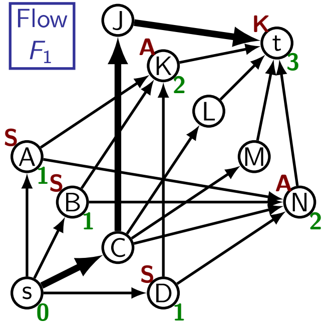
With all edge capacities 1, edge flows are either 0 or 1. Notation will be simplified:

- Edge capacities not marked.
- (Non-zero) Edge flows marked by edge thickening.
- Potential flow values not indicated on labels.

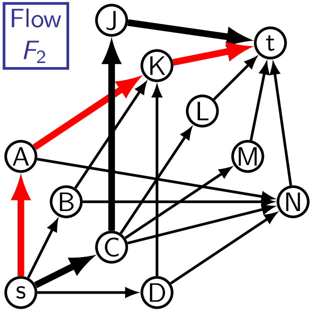


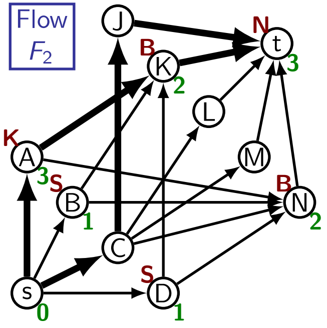
$\Rightarrow$



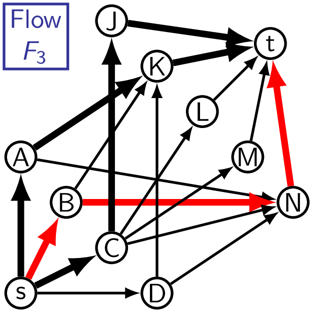


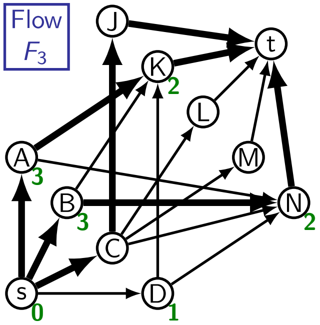
$\Rightarrow$





$\Rightarrow$





$\Rightarrow$

Even using virtual capacities to reach vertices of level 3, it isn't possible to assign a level to t, so the algorithm terminates.

The matchings are indicated by the edges between $\{A, B, C, D\}$ and $\{J, K, L, M, N\}$ that have flow 1 (the thick edges).

So the matching is

$m = \{(A, K), (B, N), (C, J)\}$.

As remarked earlier, the above solution is very easy to get without the use of the vertex labelling algorithm. We have just matched the first member of T with the first 'available' member of S, then the next member of T with the next 'available' member of S and so on.

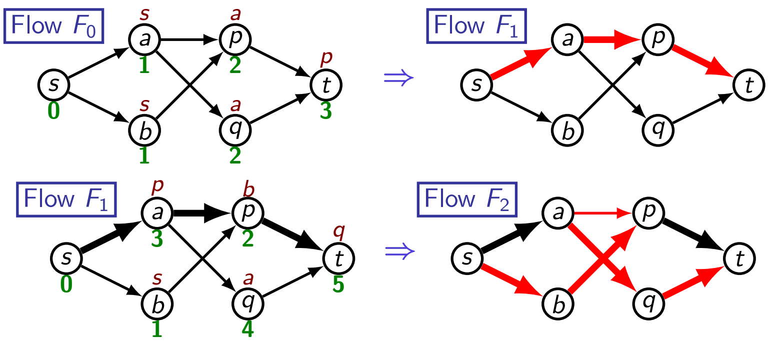
The benefit of using the algorithm lies in its ability to 'undo' initial pairings and replace them by new ones when this becomes necessary to enable later pairings.

The final example shows how it does this in the simplest possible case.

Vertex labelling for matching; Example 2

Find a (maximal) matching function m
for the relation $R = \{(a,p),(a,q),(b,p)\}$.

Obviously the only answer is $m = \{(a,q),(b,p)\}$, but the vertex labelling algorithm will first match a with p. However it will then use a virtual flow to undo this and match a with q, allowing b to be matched with p instead. Here are the diagrams:



END OF SECTION D2