We are training to be algorithms chefs, not algorithms cooks

dynamic programming
greedy algorithms
translation strategy

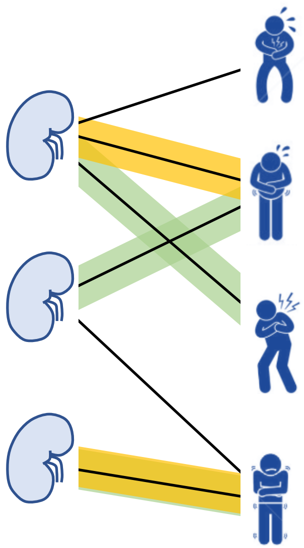
depth-first search
breadth-first search
Dijkstra's algorithm
Bellman-Ford algorithm
Johnson's algorithm

Ford-Fulkerson algorithm
matchings
topological sort
Prim's algorithm
Kruskal's algorithm

frontier
edge relaxation

SECTION 6.4

Matchings

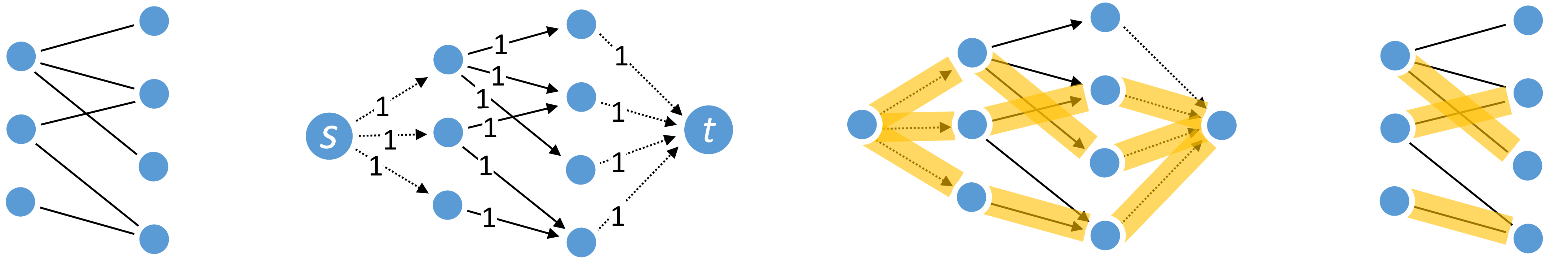


DEFINITIONS

- A **bipartite graph** is an undirected graph in which the vertices are split into two sets, and all edges go between these sets
- A **matching** in a bipartite graph is a selection of edges, such that no vertex is connected to more than one of the edges
- The **size** of a matching is the number of edges it includes
- A **maximum matching** is one with the largest possible size

PROBLEM STATEMENT

Given a bipartite graph, find a maximum matching



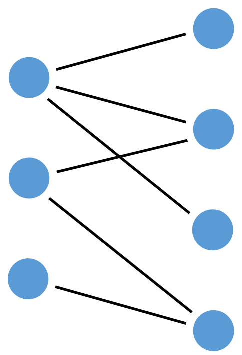
0. Given a bipartite graph ...

1. Build a helper graph:
- add source s and sink t
- add edges from s and to t

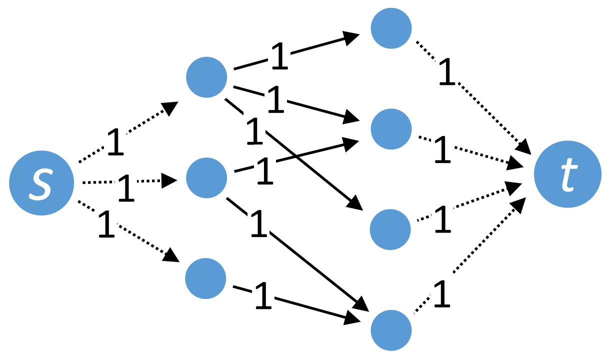
2. Solve max-flow on the helper graph, to find a maximum flow f^*

3. Interpret the flow f^* as a matching

What's the bug in my thinking?

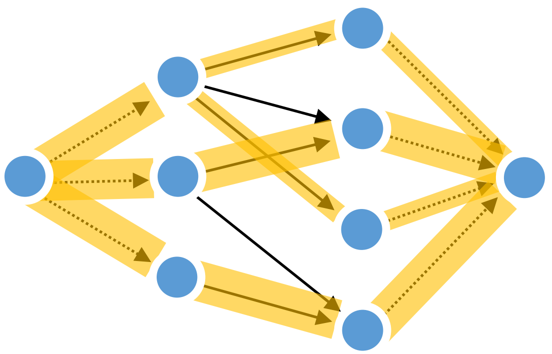


0. Given a bipartite graph ...



1. Build a helper graph:

- add source s and sink t
- add edges from s and to t

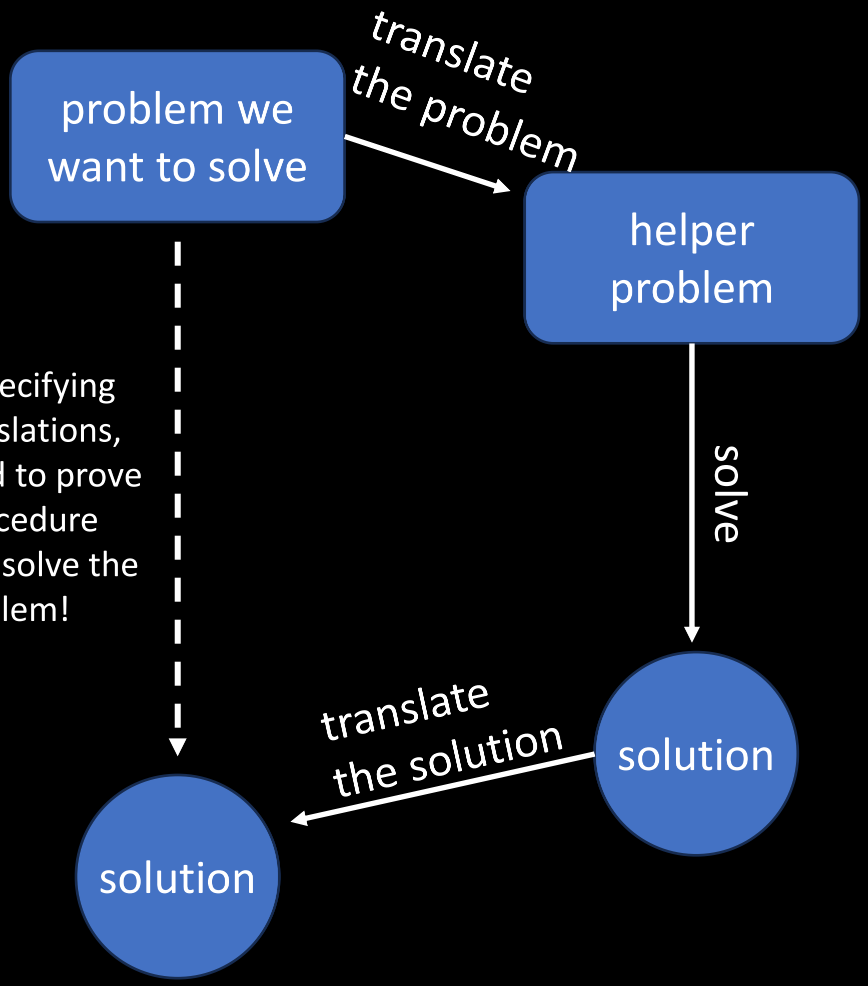


2. Solve max-flow on the helper graph, to find a maximum flow f^*

wtf ?!
This isn't the sort of flow I expected!

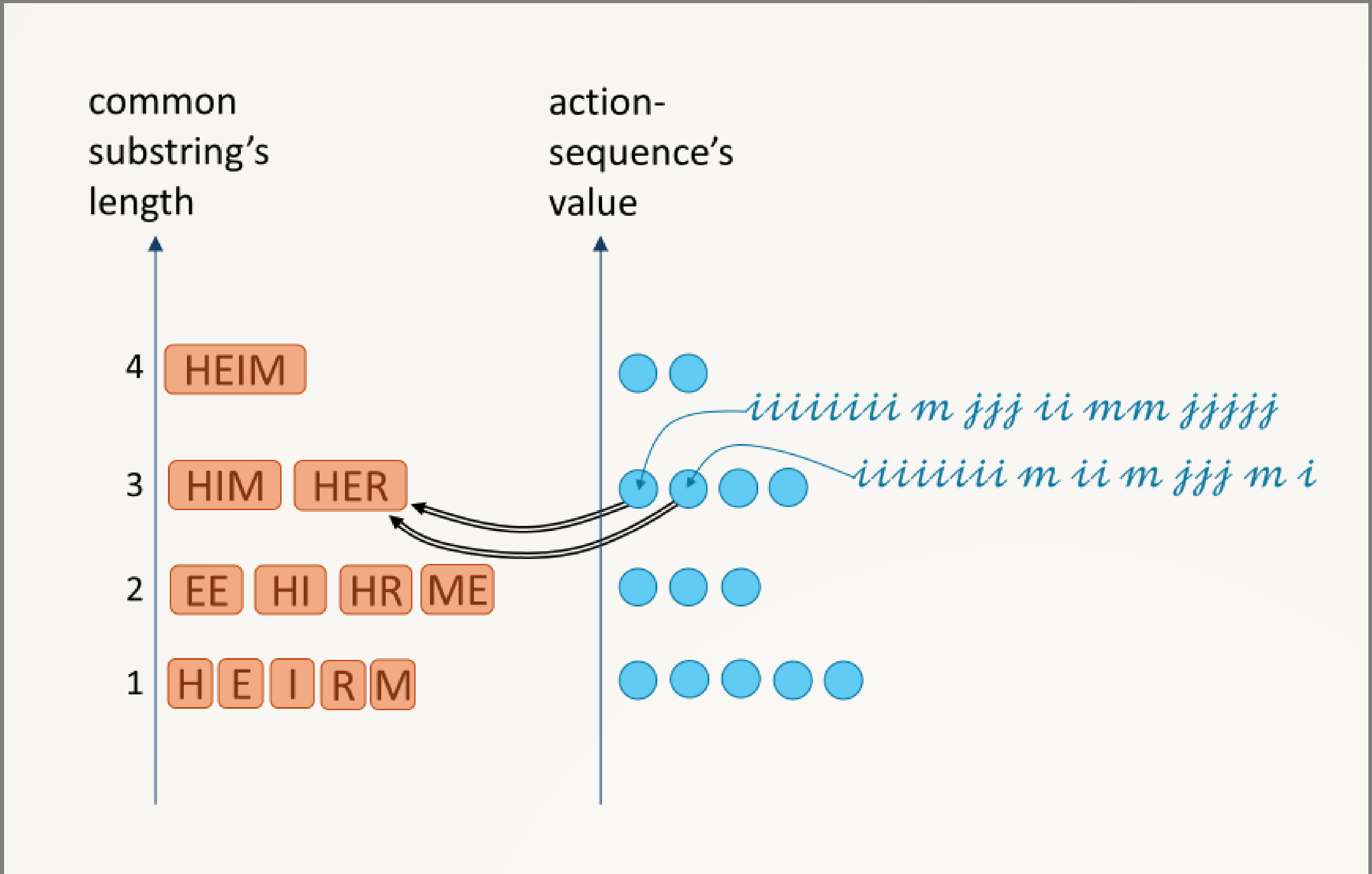
3. Interpret the flow f^* as a matching

The “Translation” strategy

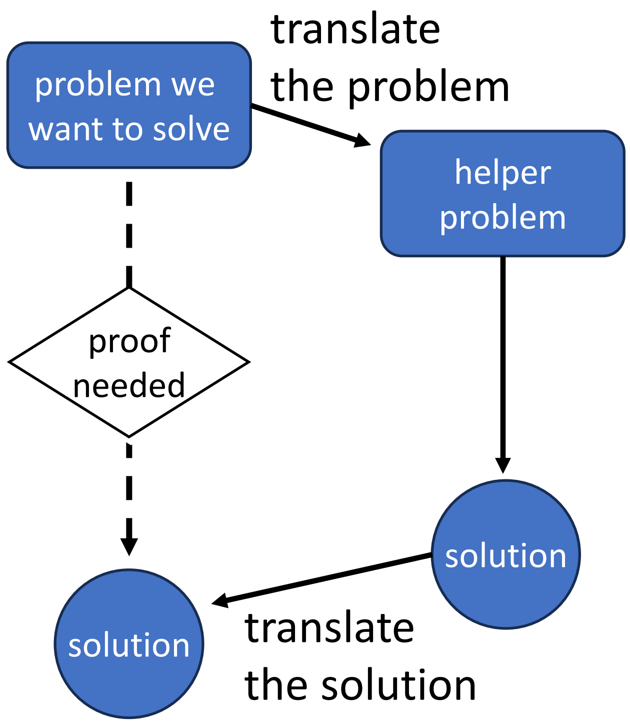


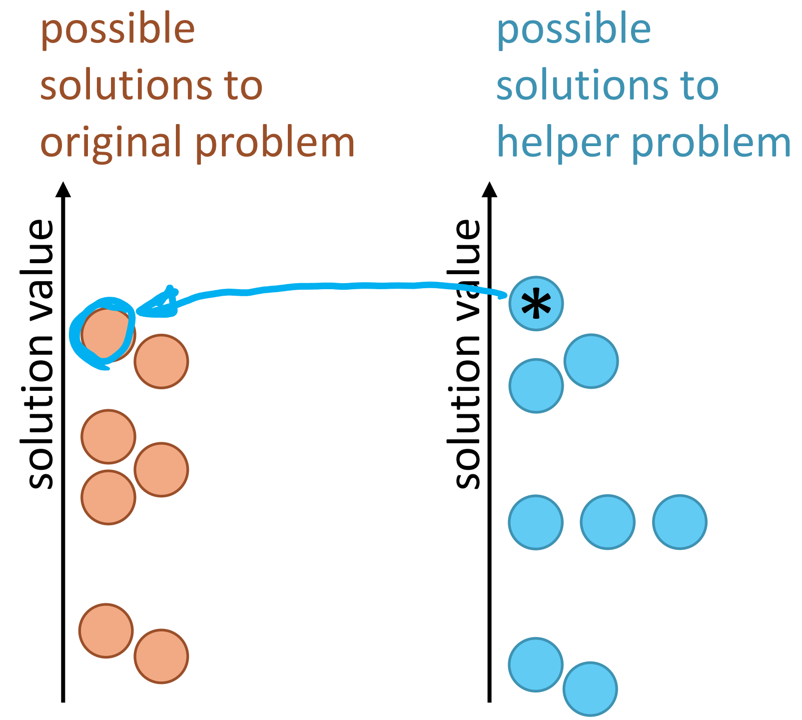
As well as specifying the two translations, we also need to prove that this procedure does indeed solve the original problem!

We used the translation strategy for finding the longest common substring using dynamic programming.



There's a common pattern when applying the translation strategy to optimization problems.





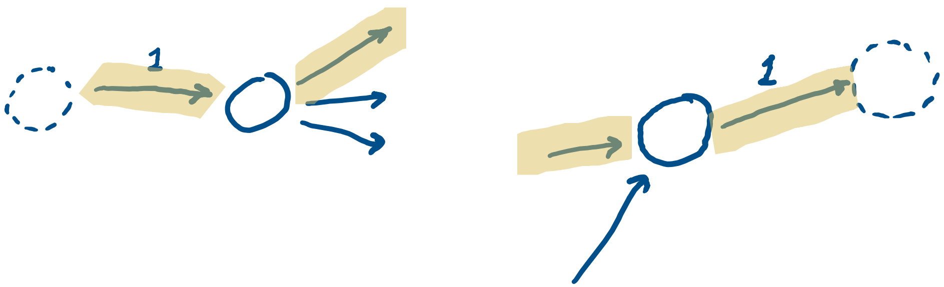
The typical way we prove correctness is ...

CLAIM1. The optimal helper solution *does* translate into a possible solution to the original problem

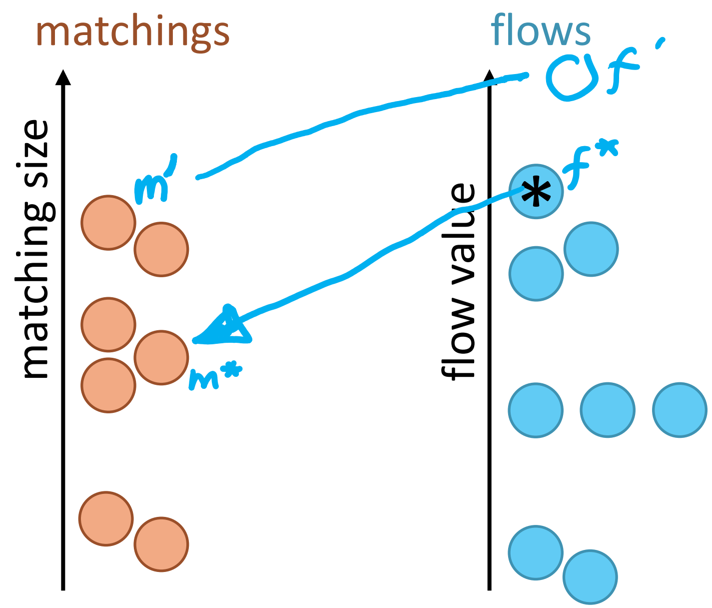
CLAIM2. This translation is optimal for the original problem

CLAIM1. The optimal helper solution *does* translate into a possible solution to the original problem

Ford-Fulkerson will produce an integer flow, since all capacities are integer. Indeed, the flow on each edge must be either 0 or 1:



Thus, the capacity constraints tell us that, when we translate f^* into an edge selection, it meets the definition of "matching".



CLAIM2. This translation is optimal for the original problem

Suppose not, i.e. suppose the max flow f^* translates to a matching m^*, but there exists a larger-size matching m'.

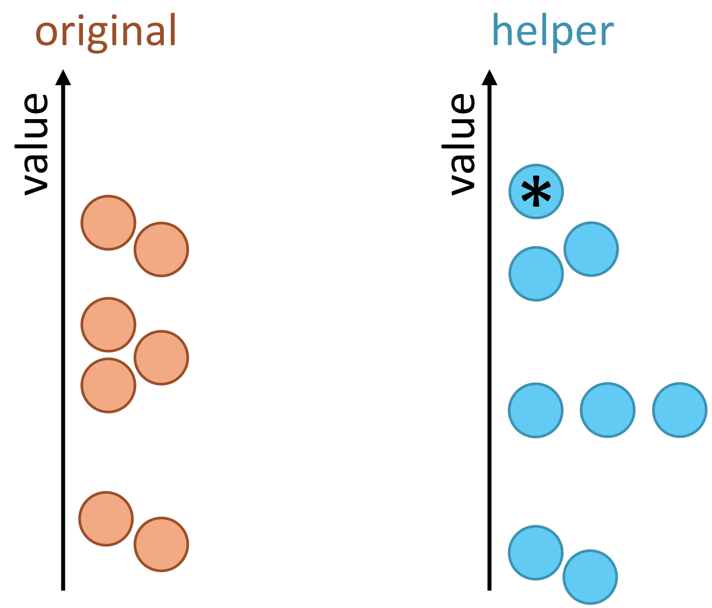
Note that when we translate matching ↔ flow in the obvious way,

value(flow) = size(matching)

Since size(m') > size(m^*)*, there is a flow f' whose value is strictly greater than the value of f^*. But this contradicts optimality of f^*.*

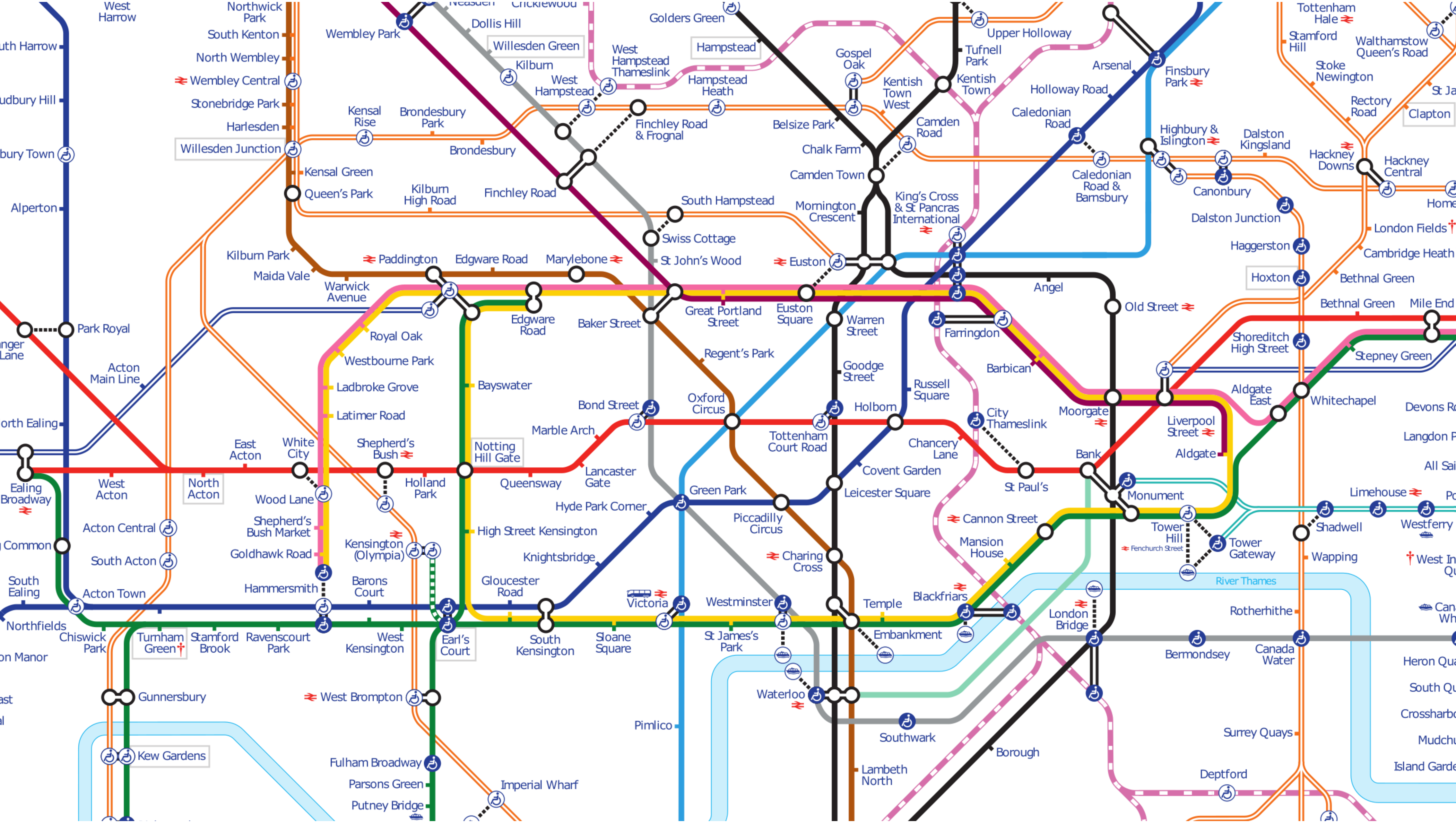
CLAIM1. The optimal helper solution *does* translate into a possible solution to the original problem

CLAIM2. This translation is optimal for the original problem

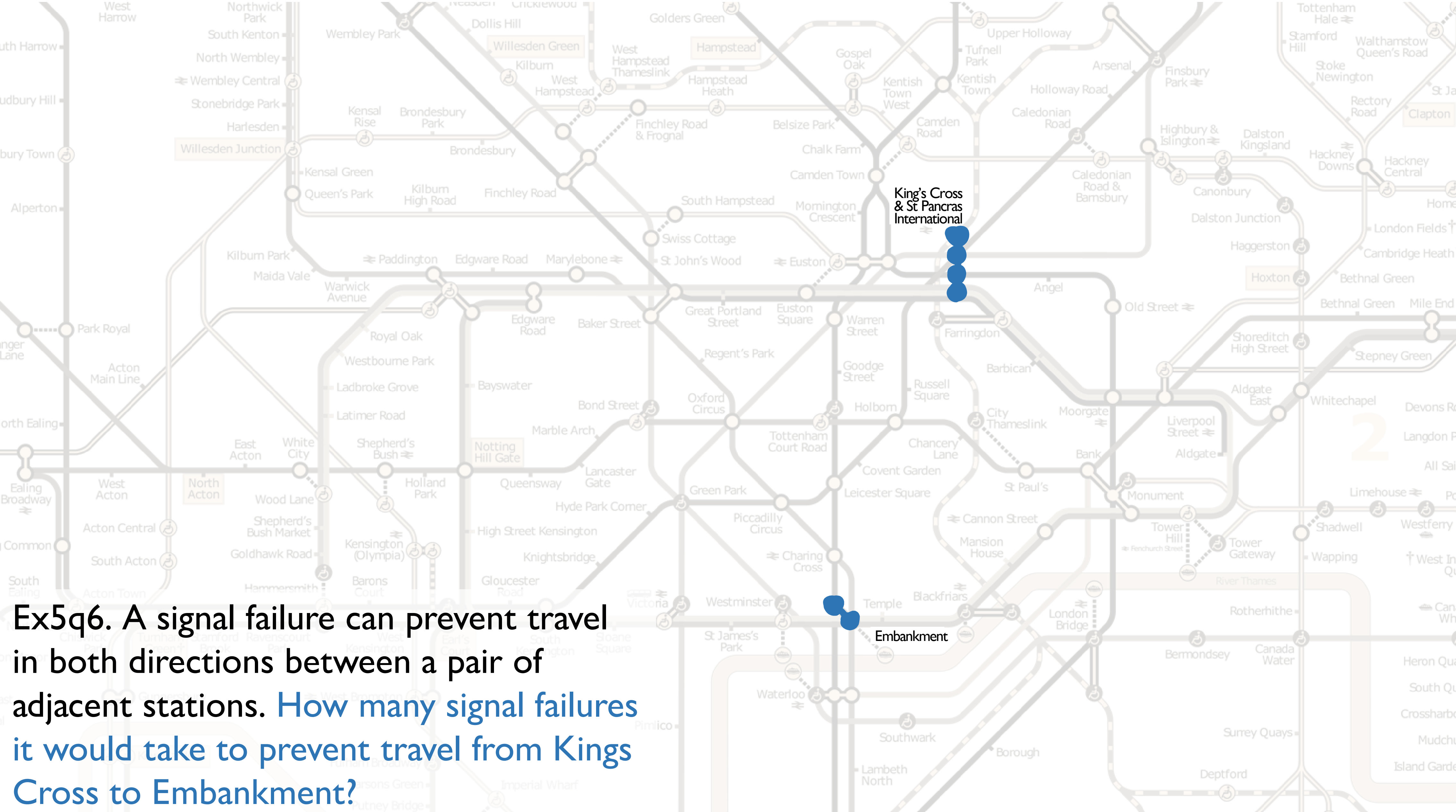


For every problem where you propose using a "Translation" strategy, you have to

- invent the two translations (original problem → helper problem, helper solution → original solution)
- prove that your translations satisfy these two claims



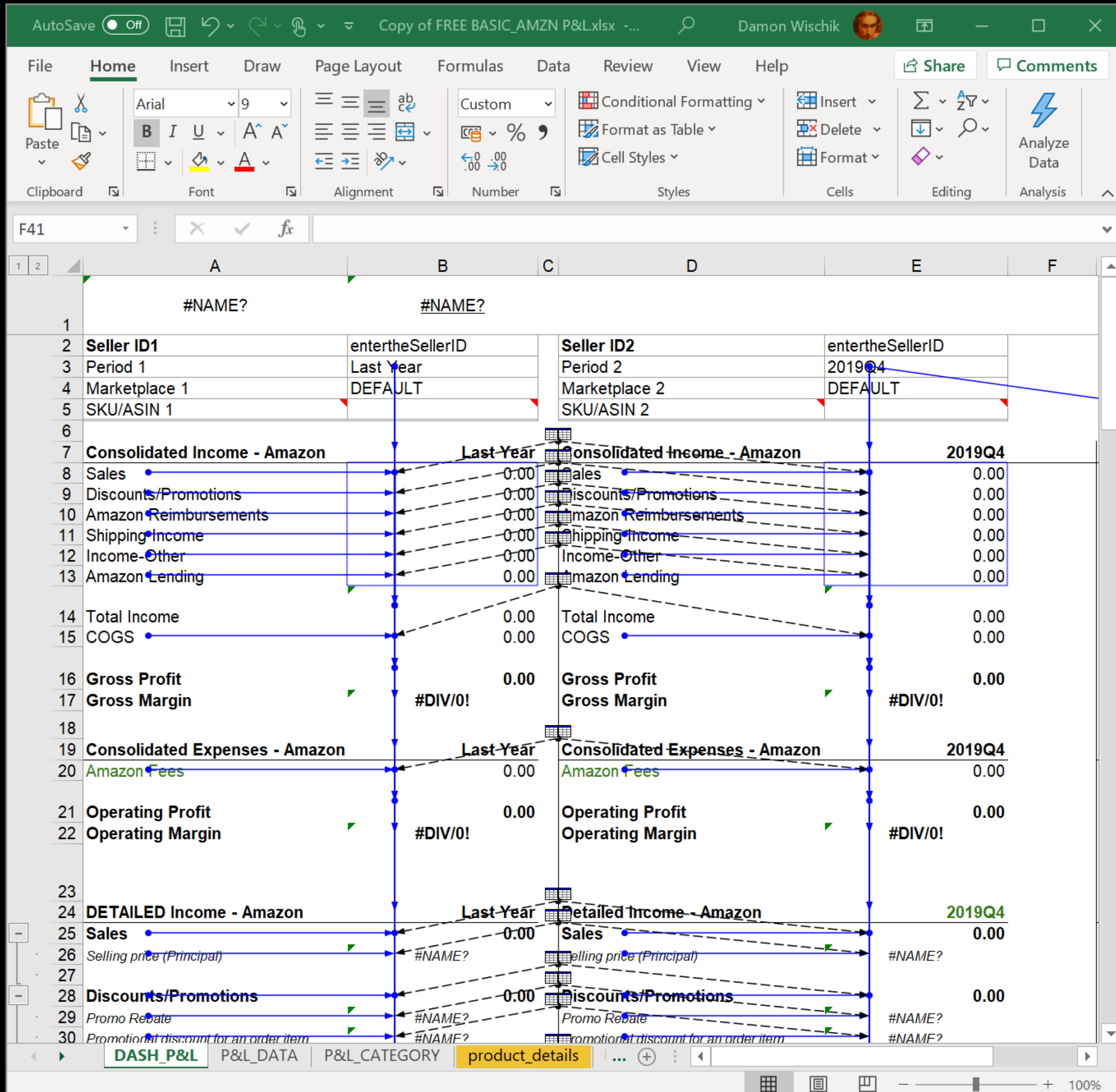
West Harrow
Northwick Park
South Kenton
North Wembley
Wembley Central
Stonebridge Park
Harlesden
Willesden Junction
Kensal Green
Queen's Park
Kilburn Park
Maida Vale
Warwick Avenue
Paddington
Edgware Road
Marylebone
Baker Street
Regent's Park
Oxford Circus
Wembley Park
Dollis Hill
Willesden Green
Kilburn
West Hampstead
Kensal Rise
Brondesbury Park
Brondesbury
Kilburn High Road
Finchley Road
Finchley Road & Frognal
West Hampstead Thameslink
Hampstead
Hampstead Heath
Golders Green
Belsize Park
Chalk Farm
Camden Town
Gospel Oak
Kentish Town West
Kentish Town
Camden Road
Tufnell Park
Upper Holloway
Arsenal
Holloway Road
Caledonian Road
Caledonian Road & Barnsbury
Finsbury Park
Highbury & Islington
Canonbury
Dalston Kingsland
Dalston Junction
Haggerston
Hoxton
Tottenham Hale
Stamford Hill
Walthamstow Queen's Road
Stoke Newington
Rectory Road
Clapton
Hackney Downs
Hackney Central
London Fields
Cambridge Heath
Bethnal Green
Mile End
South Hampstead
Swiss Cottage
St John's Wood
Mornington Crescent
King's Cross & St Pancras International
Euston
Euston Square
Great Portland Street
Warren Street
Goodge Street
Russell Square
Angel
Old Street
Farringdon
Barbican
Moorgate
Shoreditch High Street
Stepney Green
Aldgate East
Whitechapel
Liverpool Street
Aldgate
Park Royal
Acton Main Line
Royal Oak
Westbourne Park
Ladbroke Grove
Latimer Road
Bayswater
Bond Street
Marble Arch
Tottenham Court Road
Holborn
City Thameslink
Chancery Lane
Covent Garden
St Paul's
Bank
Monument
Devons Road
Langdon Park
All Saints
Ealing Broadway
West Acton
North Acton
East Acton
White City
Wood Lane
Shepherd's Bush
Holland Park
Notting Hill Gate
Queensway
Lancaster Gate
Hyde Park Corner
Green Park
Leicester Square
Piccadilly Circus
Cannon Street
Mansion House
Tower Hill
Fenchurch Street
Tower Gateway
Limehouse
Shadwell
Westferry
Wapping
Acton Central
South Acton
Acton Town
Shepherd's Bush Market
Goldhawk Road
Hammersmith
Kensington (Olympia)
Barons Court
High Street Kensington
Knightsbridge
Gloucester Road
Victoria
Westminster
Charing Cross
Temple
Blackfriars
London Bridge
River Thames
Rotherhithe
South Ealing
Northfields
Chiswick Park
Turnham Green
Stamford Brook
Ravenscourt Park
West Kensington
Earl's Court
South Kensington
Sloane Square
St James's Park
Embankment
Bermondsey
Canada Water
Heron Quays
South Quay
Crossharbour
Mudchute
Island Gardens
Gunnersbury
West Brompton
Waterloo
Southwark
Pimlico
Kew Gardens
Fulham Broadway
Parsons Green
Putney Bridge
Imperial Wharf
Lambeth North
Borough
Surrey Quays
Deptford



Ex5q6. A signal failure can prevent travel in both directions between a pair of adjacent stations. How many signal failures it would take to prevent travel from Kings Cross to Embankment?

SECTION 6.7

Topological sort

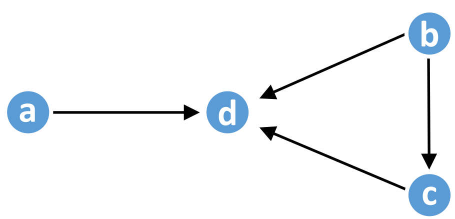

AutoSave Off
Copy of FREE BASIC_AMZN P&L.xlsx -...
Damon Wischik
File
Home
Insert
Draw
Page Layout
Formulas
Data
Review
View
Help
Share
Comments
Paste
Arial
9
Custom
Conditional Formatting
Format as Table
Cell Styles
Insert
Delete
Format
Analyze Data
Clipboard
Font
Alignment
Number
Styles
Cells
Editing
Analysis
F41
#NAME?
#NAME?
Seller ID1
entertheSellerID
Seller ID2
entertheSellerID
Period 1
Last Year
Period 2
2019Q4
Marketplace 1
DEFAULT
Marketplace 2
DEFAULT
SKU/ASIN 1
SKU/ASIN 2
Consolidated Income - Amazon
Last Year
Consolidated Income - Amazon
2019Q4
Sales 0.00
Discounts/Promotions 0.00
Amazon Reimbursements 0.00
Shipping Income 0.00
Income-Other 0.00
Amazon Lending 0.00
Total Income 0.00
COGS 0.00
Gross Profit 0.00
Gross Margin #DIV/0!
Consolidated Expenses - Amazon
Last Year
2019Q4
Amazon Fees 0.00
Operating Profit 0.00
Operating Margin #DIV/0!
DETAILED Income - Amazon
Last Year
Detailed Income - Amazon
2019Q4
Sales 0.00
Selling price (Principal) #NAME?
Discounts/Promotions 0.00
Promo Rebate #NAME?
Promotional discount for an order item #NAME?
DASH_P&L
P&L_DATA
P&L_CATEGORY
product_details
100%

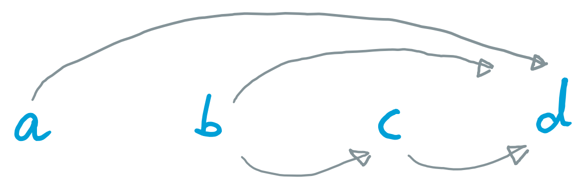
DEFINITION

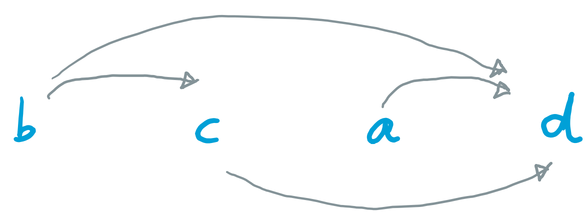
Given a directed graph, a **total ordering** is an ordering of the vertices such that if there is an edge $v \to u$ in the graph, then $v < u$ in the ordering.

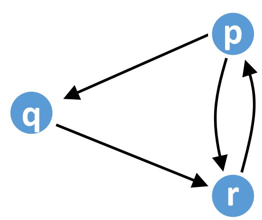
PROBLEM STATEMENT

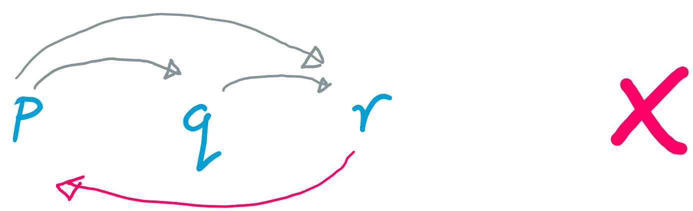
Find a total ordering, if one exists.











This graph has a cycle, so there is no total order possible.

We are training to be algorithms chefs, not algorithms cooks

dynamic programming
greedy algorithms
translation strategy

heap

depth-first search
breadth-first search
Dijkstra's algorithm
Bellman-Ford algorithm
Johnson's algorithm

Ford-Fulkerson algorithm
matchings
topological sort
Prim's algorithm
Kruskal's algorithm

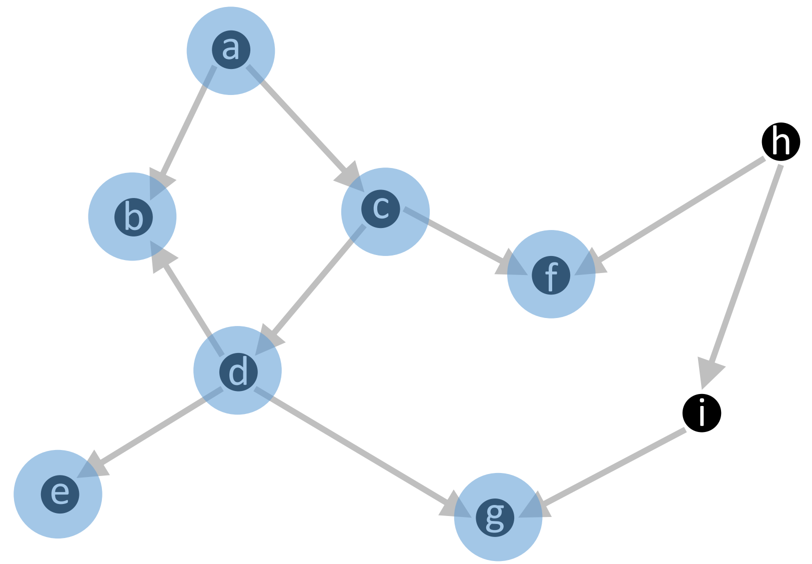
?
?
?

These are interesting ideas, worth pursuing. We'll pursue one of them: depth-first search.

```python
def dfs_recurse(g, s):
    for v in g.vertices:
        v.visited = False
    visit(s)

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

attempt 1: depth-first search

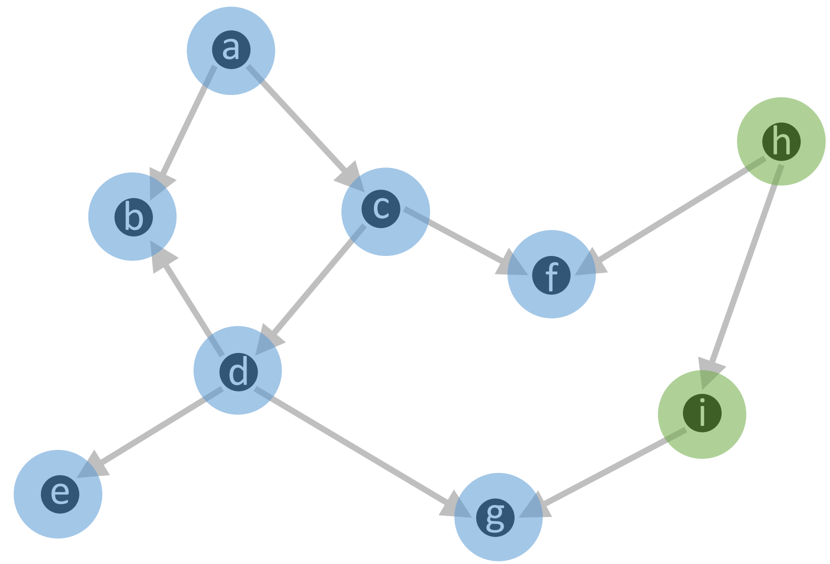


This might not even visit all vertices, so it might not produce a total order.

```python
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)

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

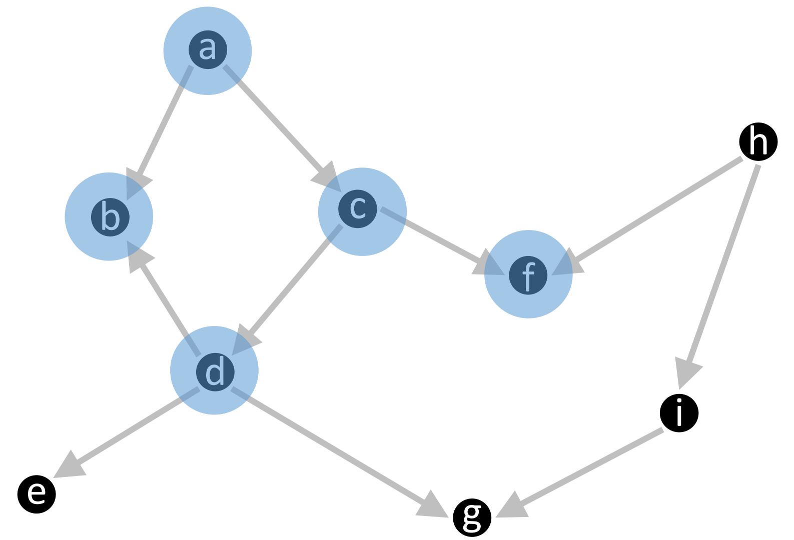
attempt 2: comprehensive depth-first search



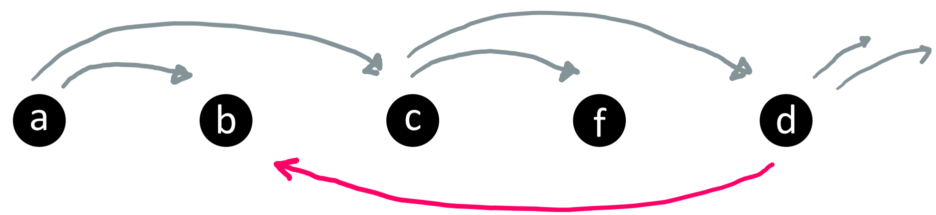
```python
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)

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

attempt 2: comprehensive depth-first search



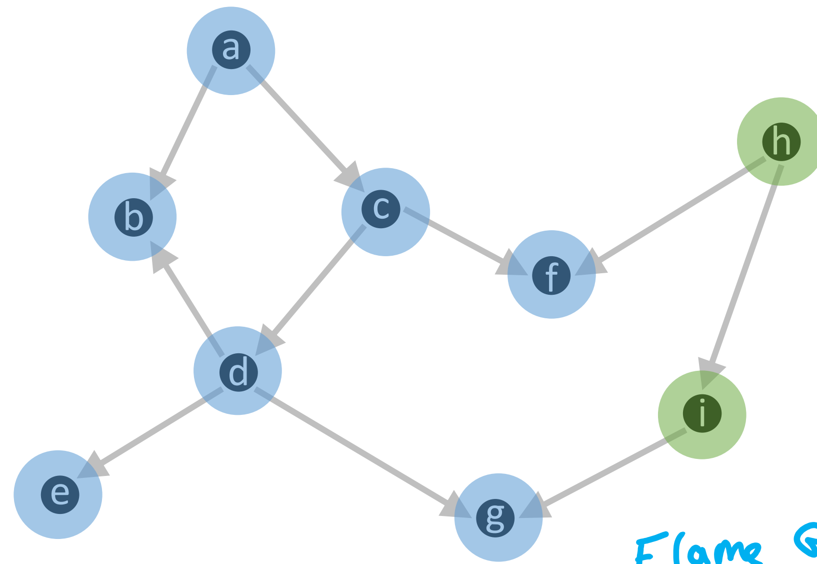
Some edges point backwards – not a total order.



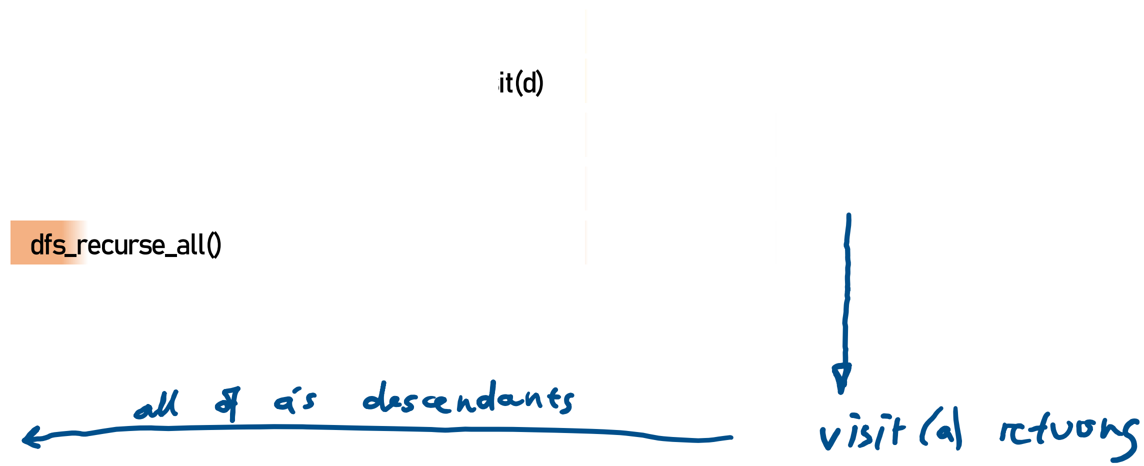
```python
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)

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

attempt 2: comprehensive depth-first search

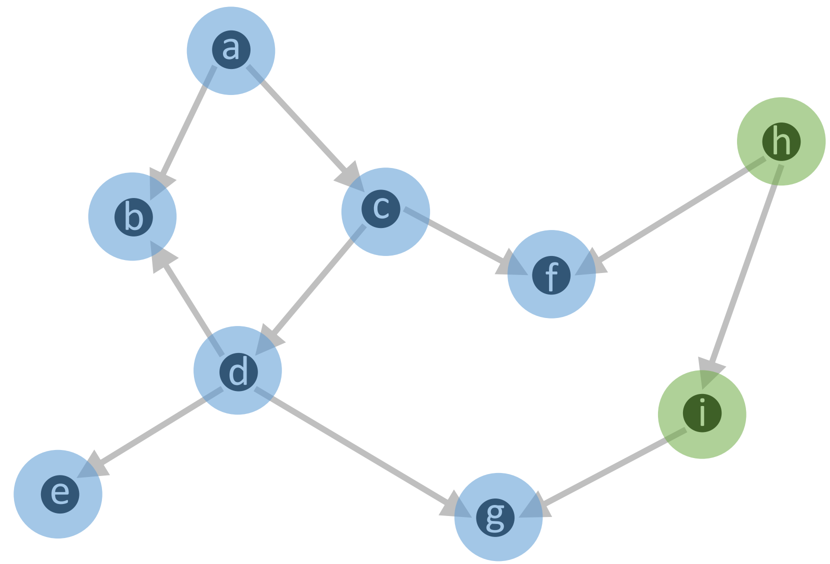


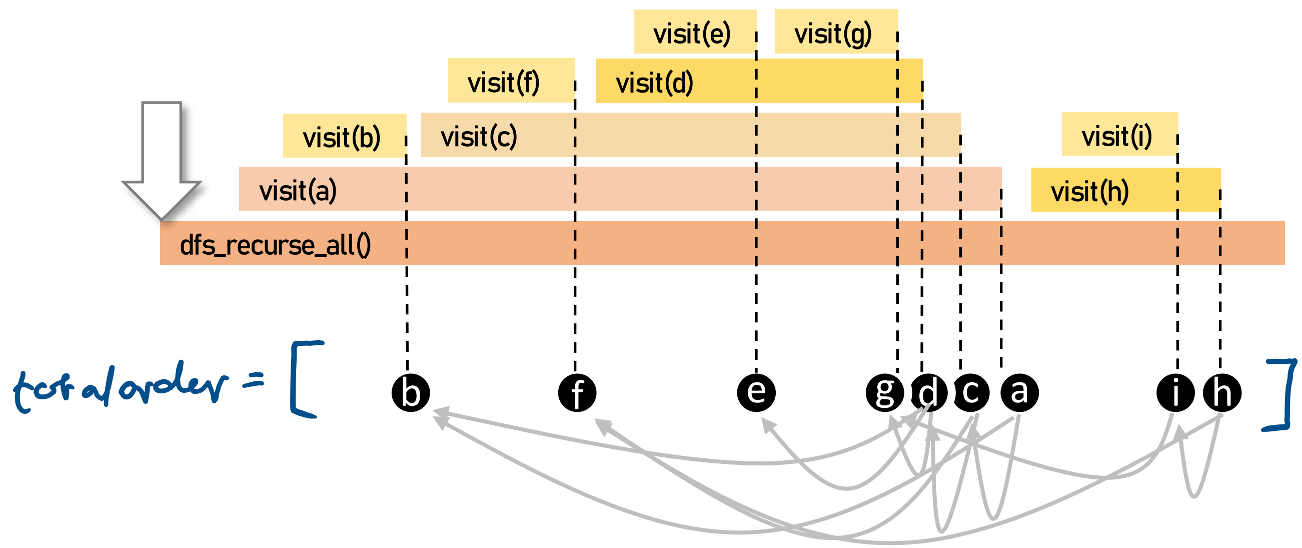
Flame Graph



```python
def toposort(g):
    for v in g.vertices:
        v.visited = False
        # v.colour = 'white'
    totalorder = []
    for v in g.vertices:
        if not v.visited:
            visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True
    # v.colour = 'grey'
    for w in v.neighbours:
        if not w.visited:
            visit(w, totalorder)
    totalorder.append(v)
    # v.colour = 'black'
```





```
def toposort(g):
    for v in g.vertices:
        v.visited = False
        # v.colour = ‘white’
    totalorder = []
    for v in g.vertices:
        if not v.visited:
            visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True
    # v.colour = ‘grey’
    for w in v.neighbours:
        if not w.visited:
            visit(w, totalorder)
    totalorder.append(v)
    # v.colour = ‘black’
```

Correctness theorem.

Given a DAG g, this algorithm produces a `totalorder` such that for every edge $v_1 \to v_2$, v_1 appears to the right of v_2 in `totalorder`.

Performance analysis.

It has running time $O(V+E)$, just like depth-first search.

DAG = directed acyclic graph.

We’ve already seen that if there *are* cycles then it’s impossible for there to be a total order.

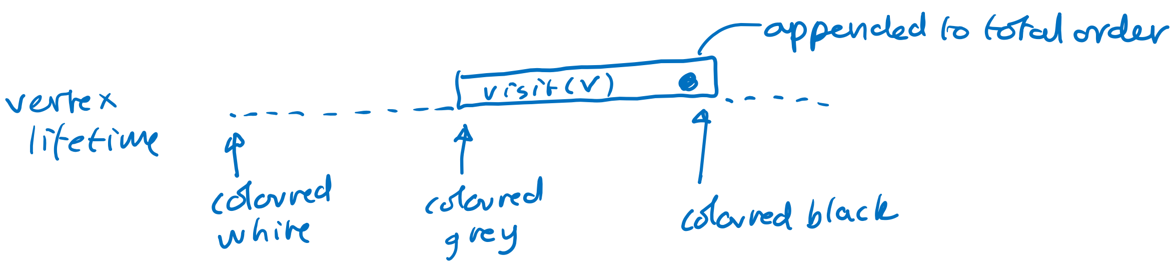
The theorem tells us that the converse is also true: if there *aren’t* any cycles then ∃ a total order.

Correctness theorem. Given a DAG g, this algorithm returns a totalorder such that for every edge $v_1 \to v_2$, totalorder has $[\cdots v_2 \cdots v_1 \cdots]$.

Proof First, the algorithm must terminate (because of how it uses the 'visited' flag.)

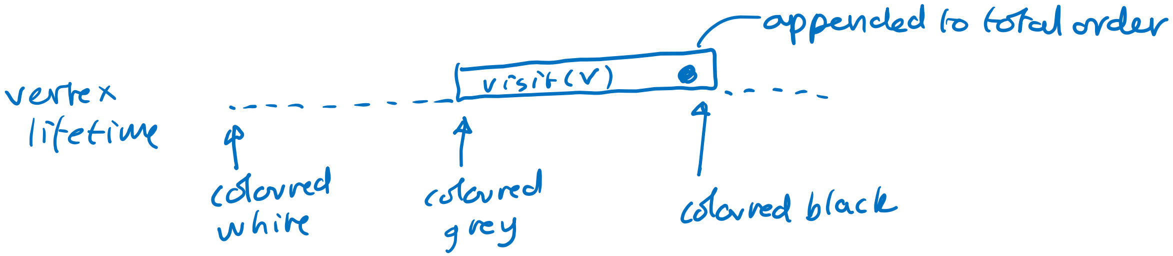
(We have to prove termination first. If it doesn't terminate, it can't return anything!)

Next, we prove the claim using the "breakpoint" strategy. We'll talk about "vertex colours", as set in the comments of the code. These colours are a way to express "what has happened in the past" in terms of "colours of the vertices right now". It's just to save us some circumlocution.



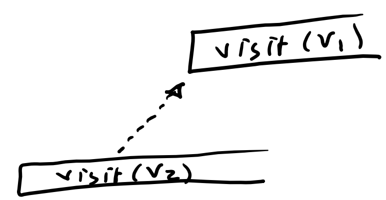

```
def toposort(g):
    for v in g.vertices:
        v.visited = False
        # v.colour = 'white'
    totalorder = []
    for v in g.vertices:
        if not v.visited:
            visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True
    # v.colour = 'grey'
    for w in v.neighbours:
        if not w.visited:
            visit(w, totalorder)
    totalorder.append(v)
    # v.colour = 'black'
```



Pick an arbitrary edge $v_1 \rightarrow v_2$, and consider the instant that v_1 got coloured grey. (It must happen at some point in execution). What colour is v_2?

- If v_2 is black: then v_2 is already in total order, so $v_2 < v_1$ ✓.
- If v_2 is white: then v_2 has not yet been visited. It's a descendent of v_1, so visit(v_2) will be invoked and terminate before visit(v_1) terminates, so $v_2 < v_1$ in total order ✓.
- If v_2 is grey, then visit(v_2) has started but not yet terminated. Therefore v_1 must be a descendant of v_2 (and the flame graph tells us a path $v_2 \rightsquigarrow v_1$).



But $v_1 \rightarrow v_2$ by assumption, hence there's a cycle, which contradicts our DAG assumption. ✗

An alternative approach to finding a total order

Preorders

Definition 139 *A* preorder $(P, \sqsubseteq)$ *consists of a set* P *and a relation* $\sqsubseteq$ *on* P *(i.e.* $\sqsubseteq \in \mathcal{P}(P \times P)$*) satisfying the following two axioms.*

- *Reflexivity.*

$$\forall x \in P.\ x \sqsubseteq x$$

- *Transitivity.*

$$\forall x, y, z \in P.\ (x \sqsubseteq y \land y \sqsubseteq z) \implies x \sqsubseteq z$$

Definition 140 *A* partial order*, or* poset[a]*, is a preorder* $(P, \sqsubseteq)$ *that further satisfies*

- *Antisymmetry.*

$$\forall x, y \in P.\ (x \sqsubseteq y \land y \sqsubseteq x) \implies x = y$$

[a] (standing for *partially ordered set*)

Theorem 141 *For* $R \subseteq A \times A$*, let*

$$\mathcal{F}_R = \{ Q \subseteq A \times A \mid R \subseteq Q \land Q \text{ is a preorder} \} .$$

Then, (i) $R^{\circ *} \in \mathcal{F}_R$ *and* (ii) $R^{\circ *} \subseteq \bigcap \mathcal{F}_R$*. Hence,* $R^{\circ *} = \bigcap \mathcal{F}_R$.

Let $x \sqsubseteq y$ mean "y depends on x".
This is a partial order (and the theorem explains why partial orders correspond to directed acyclic graphs).

Might this lead to an efficient algorithm? If we have V vertices ie items to be sorted, and E edges ie relations,

- sorting algorithms are $O(V^2)$ or $O(V \log V)$
- DFS-based toposort is $O(V + E)$
- $E \leq V^2$

So, on highly connected graphs, sorting algorithms might do better.

IDEA. Think through all our sorting algorithms, and see if they can be adapted to work with partial orders.