Lecture 8

COMP 3760

Solving problems with graph algorithms

Topological Sorting (Text chapter 4.2)

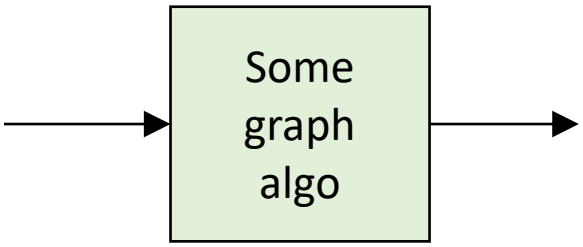
Solving problems with graph algorithms

How can we use graph algorithms to solve problems

Two strategies

1. Modify a known graph algorithm
   - Technically we have already done this
   - DFS and BFS in class notes did not perform output
   - But we tracked it in the examples

2. Use a known graph algorithm as a black box
   - Black box needs input
   - Black box gives output



- Bonus strategy:

Example: Connected components

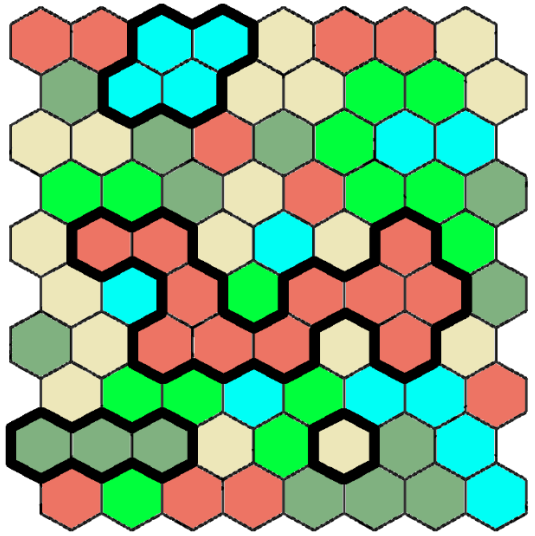
- Problem: Given a graph G, how many *connected components* does G have?

- Strategy 1: *Modify a known graph algorithm*

- Solution idea: Use either DFS or BFS
  - Add a counter to the “main loop”
  - Count how many times (from main) the helper function is called
  - Return the counter

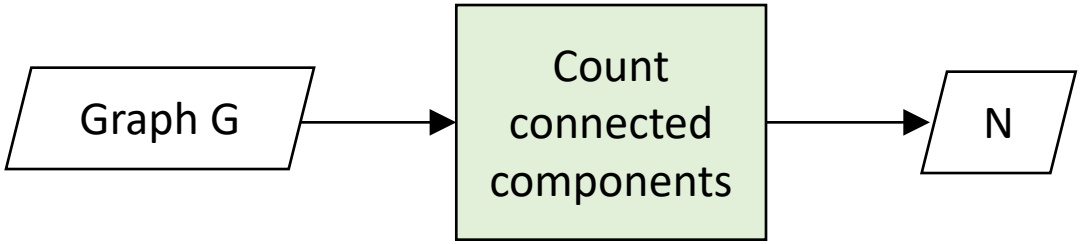
Example 2: Count map regions

- A board game is played on hex tiles.
- Tiles are drawn at random from a large supply.
- Adjacent tiles of the same colour make up a *region*.
- Input data is a list of the colours for all tiles:
  - colour(0,0) = red
  - colour(4,4) = aqua
- Problem: determine how many regions are on the map.
  - This map has 38 regions →



Solution idea

- Strategy 2: Use the “connected components” algorithm as a black box

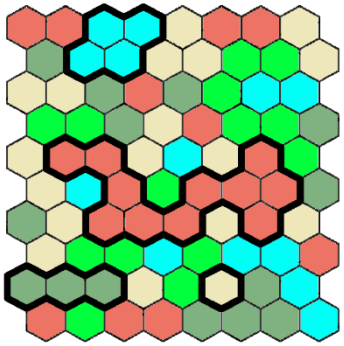


- We need to construct a clever graph – *just the right graph*
  - It will encapsulate or model or represent our input problem in some way

Finding the right graph

We are given a list of colour/point data:

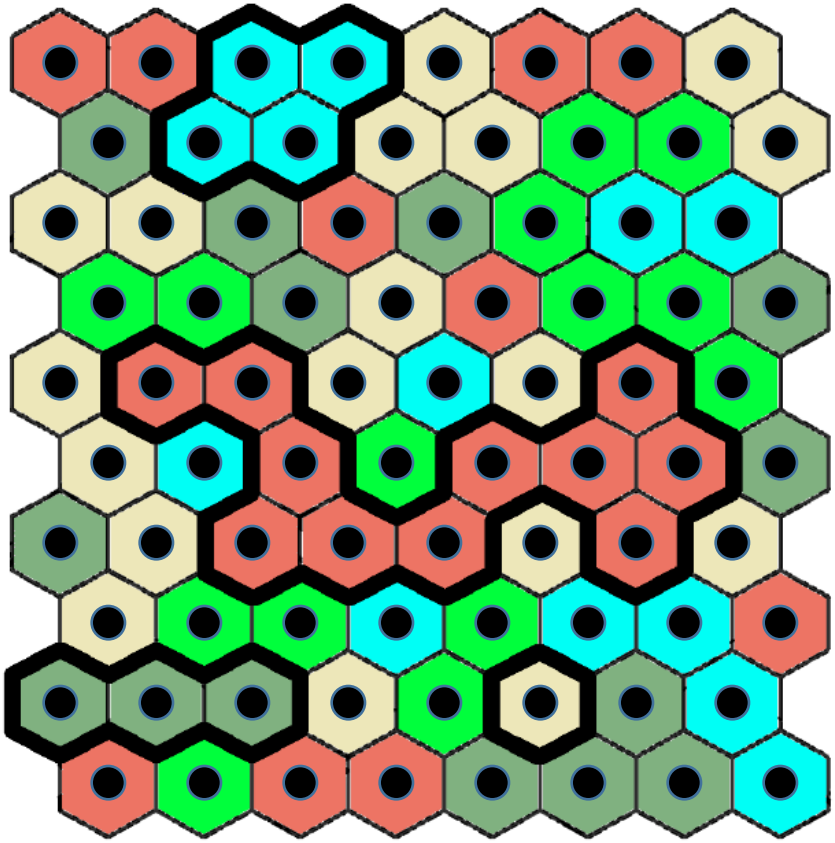
colour(0,0) = red
colour(0,1) = red
colour(0,2) = aqua
colour(0,3) = aqua
colour(0,4) = beige
...
colour(9,7) = aqua



- For a graph we need vertices and edges
- Vertices represent *things* and edges represent *relationships between things*

- The things we have are tiles
  - Idea: represent each tile as a vertex
  - Every vertex "label" will be a point (the grid location)

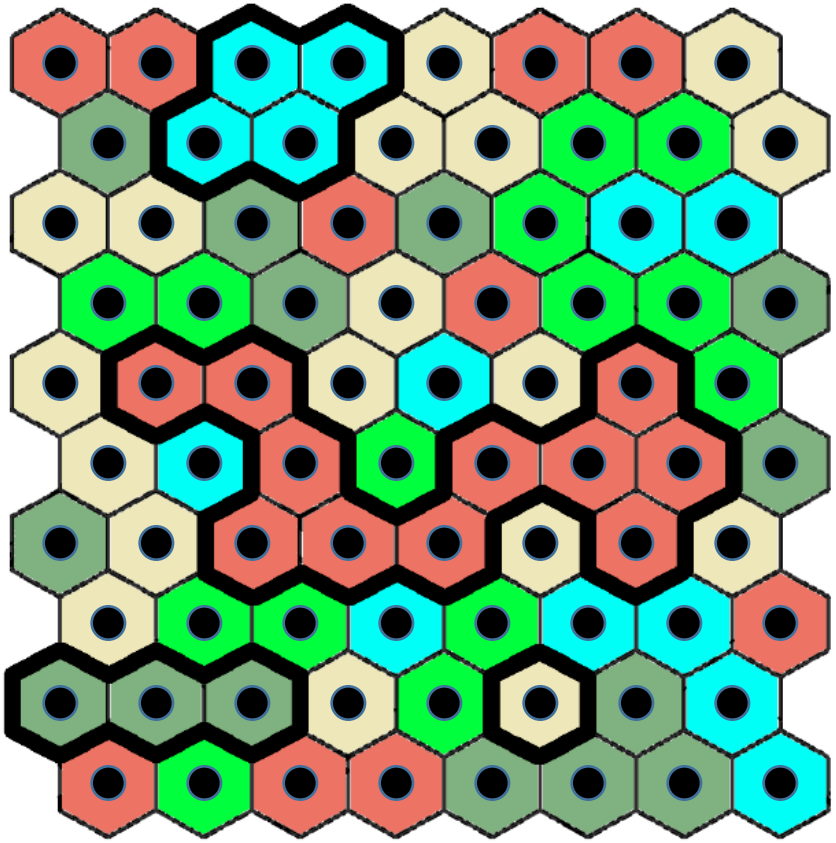
Every tile is a vertex



V = {(0,0), (0,1), (0,2), (0,3), (0,4),
(0,5), (0,6), (0,7), (1,0), (1,1), (1,2),
(1,3), (1,4), (1,5), (1,6), (1,7), (2,0),
(2,1), (2,2), (2,3), (2,4), (2,5), (2,6),
(2,7), (3,0), (3,1), (3,2), (3,3), (3,4),
(3,5), (3,6), (3,7), (4,0), (4,1), (4,2),
(4,3), (4,4), (4,5), (4,6), (4,7), (5,0),
(5,1), (5,2), (5,3), (5,4), (5,5), (5,6),
(5,7), (6,0), (6,1), (6,2), (6,3), (6,4),
(6,5), (6,6), (6,7), (7,0), (7,1), (7,2),
(7,3), (7,4), (7,5), (7,6), (7,7), (8,0),
(8,1), (8,2), (8,3), (8,4), (8,5), (8,6),
(8,7), (9,0), (9,1), (9,2), (9,3), (9,4),
(9,5), (9,6), (9,7) }

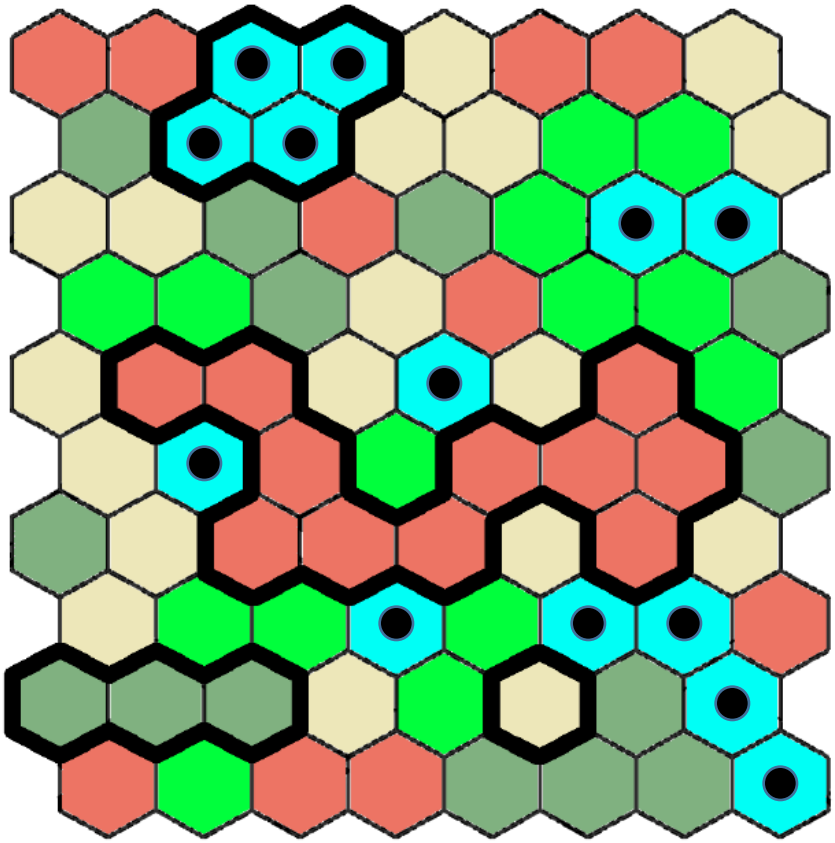
Now how about edges?

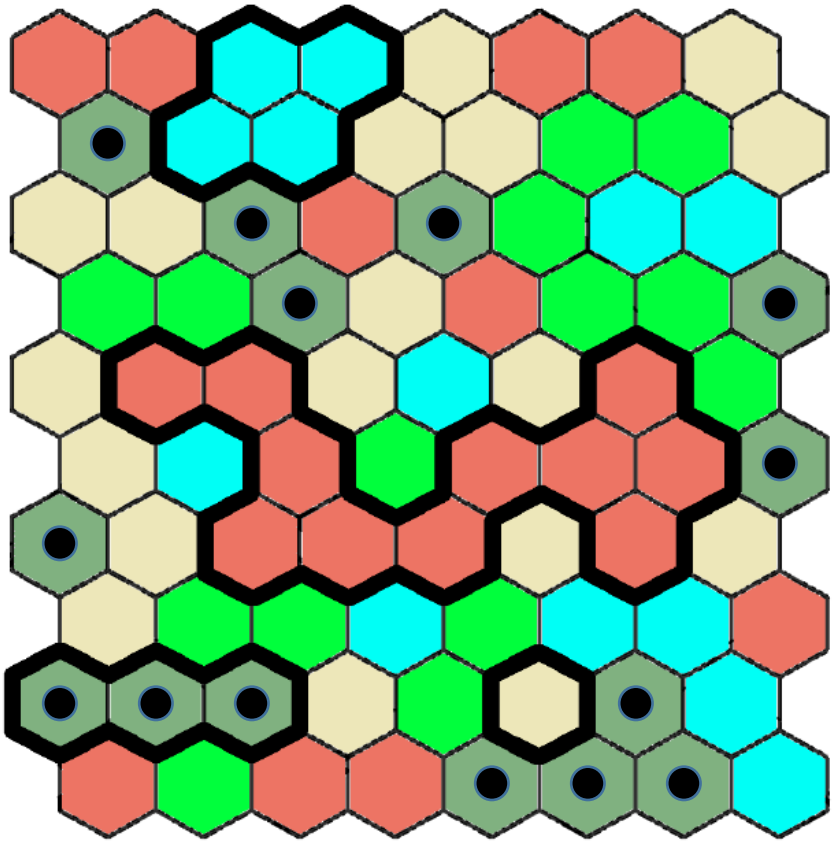
- Idea: Connect vertices of the same colour

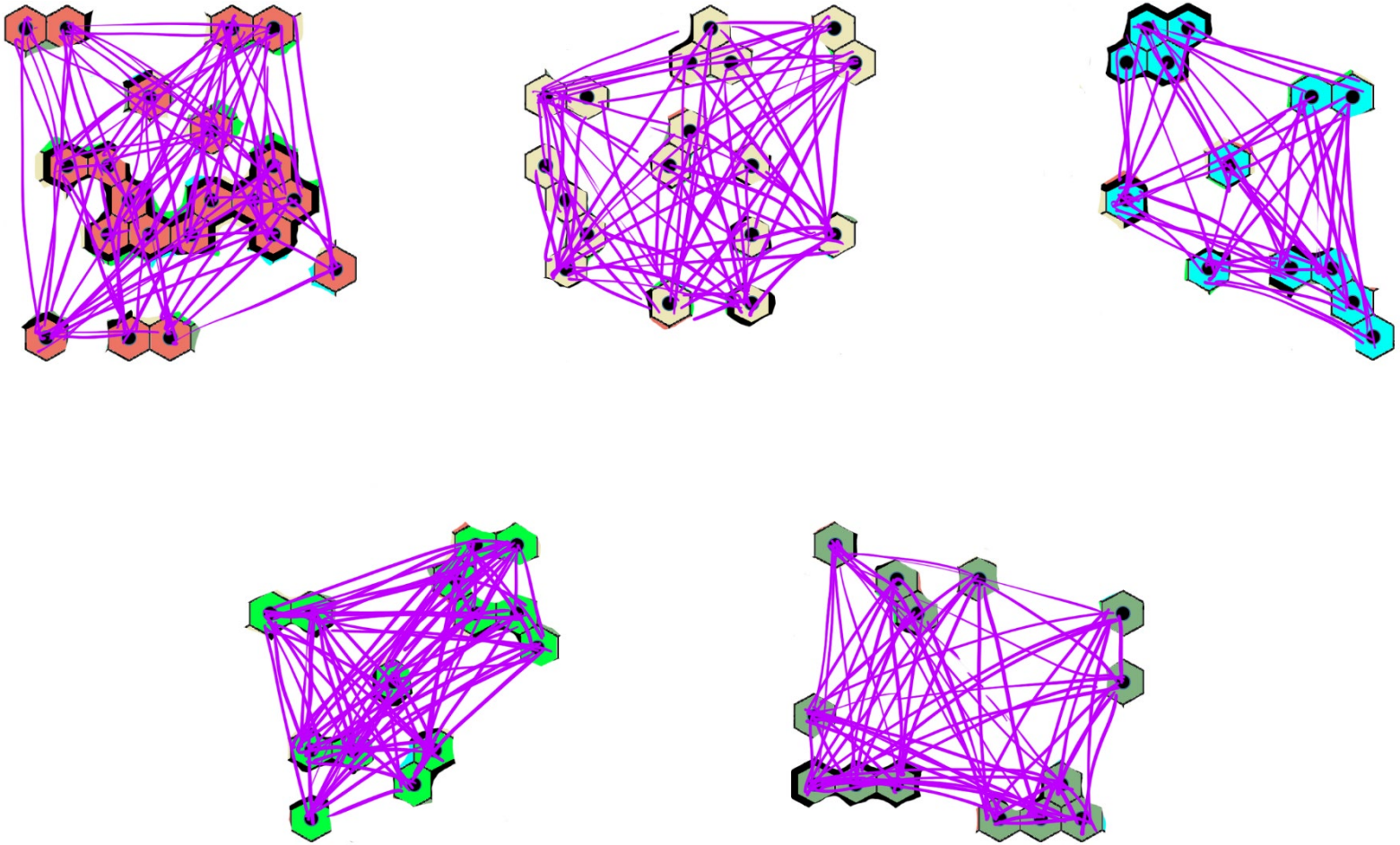


Now how about edges?

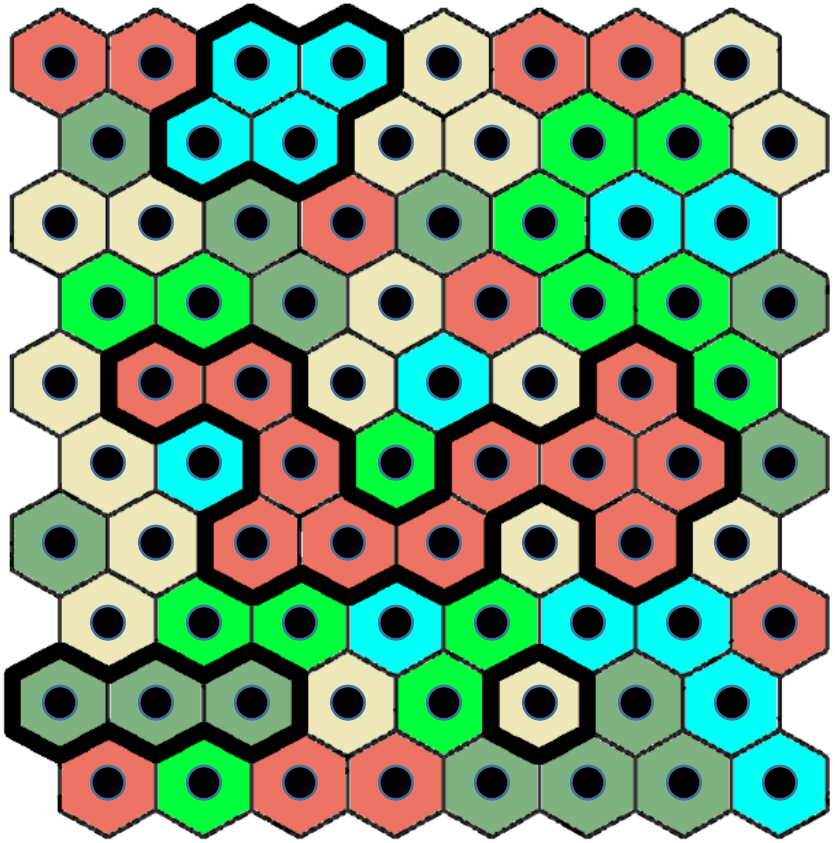
- Idea: Connect vertices of the same colour



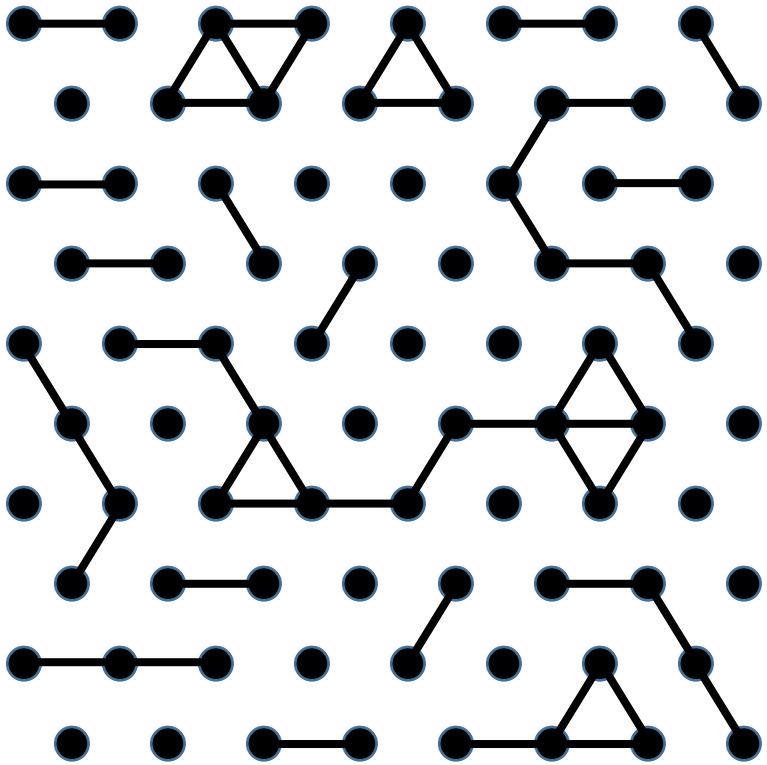




Idea 2: Same colour AND adjacent



Idea 2: Same colour AND adjacent



The Solution

- Step 1: Define a Graph G=(V, E) as follows:
  - The vertices of G are the tiles of the map; each is represented by a grid location aka point (0,0) to (m,n)
  - There is an edge between two vertices u and v iff
    - u and v have the same colour AND
    - u and v are in adjacent locations
- Step 2: Run “Count Connected Components” on G
- Step 3: The output of step 2 is the final answer

Strategy 2 (using a graph algorithm as a black box)

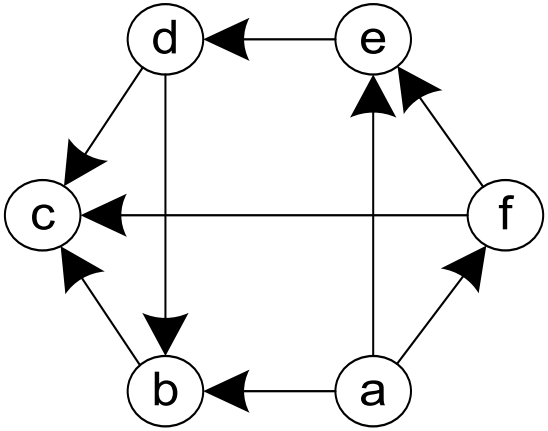
- *You MUST specify what is the input (usually a graph, maybe other info)*
- *You MUST specify how the output (of the graph algorithm) determines the answer to the problem*

Graph Algorithms: Topological sorting

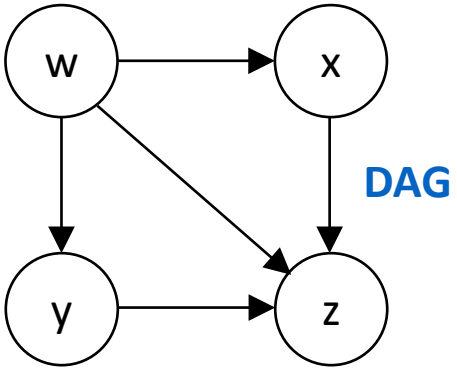
Textbook: Chapter 4.2

Directed acyclic graphs (DAGs)

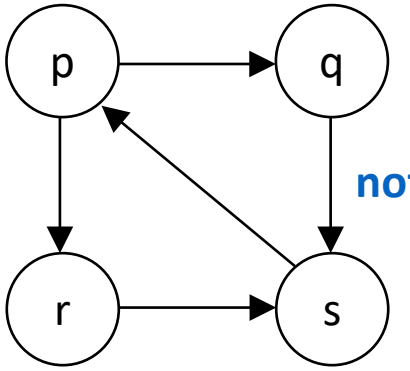
- A **directed graph** is a graph whose edges are directional or one-way



- A **directed acyclic graph** is a directed graph that contains no cycles



DAG



not a DAG

Topological sort problem

- Given a set of tasks with dependencies (precedence constraints), *e.g.*, “task A must be completed before task B”, …

- … find a linear ordering of the tasks that satisfies all dependencies

Example: Getting dressed

- Suppose you need to wear all these items:

• Belt	• Shirt	• Suspenders
• Jacket	• Shoes	• Tie
• Pants	• Socks	• Underwear

- Some of these items must come before others

Example: Getting dressed

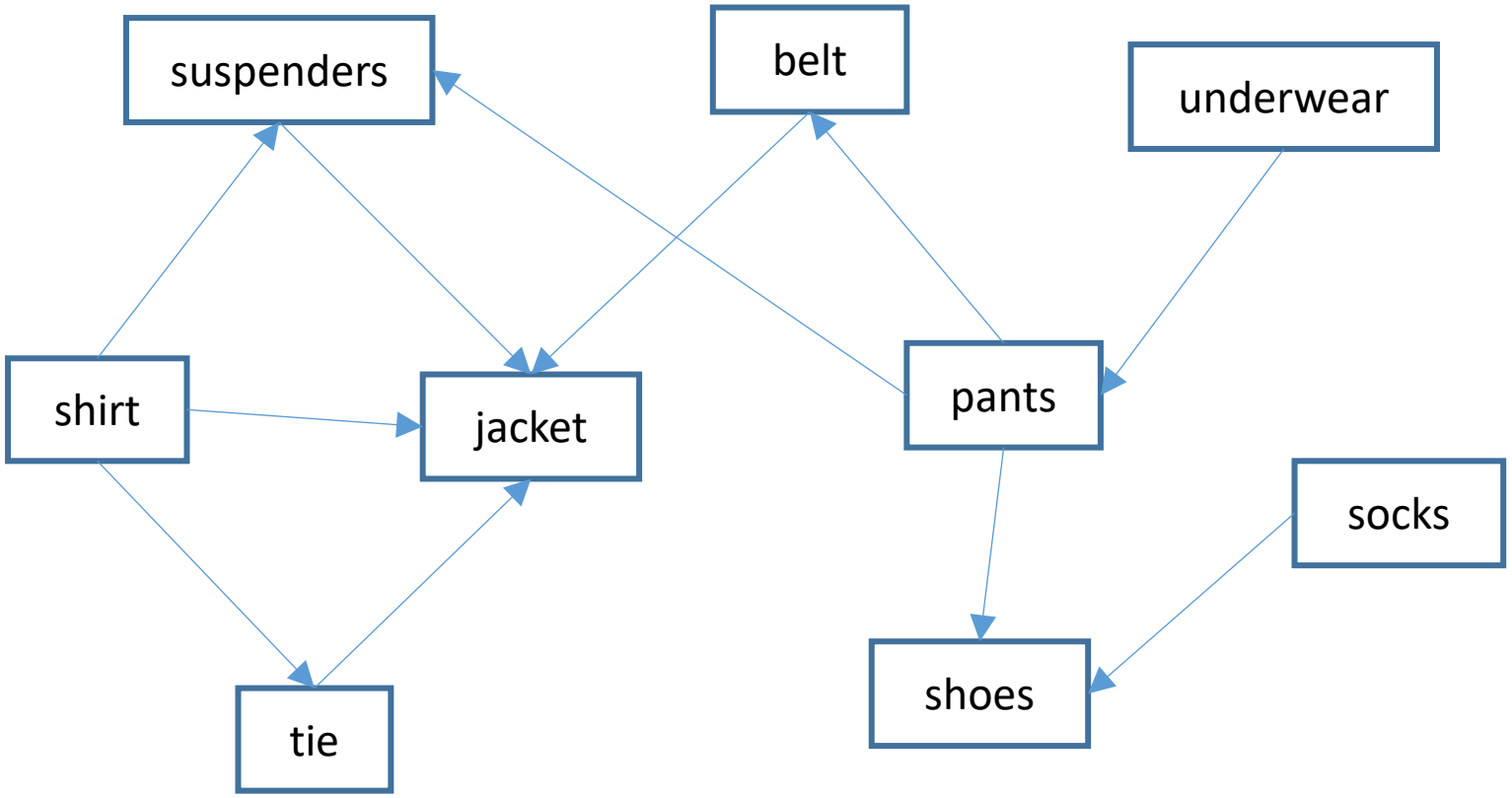
- Socks before shoes
- Shirt before suspenders
- Pants before suspenders
- Pants before shoes
- Pants before belt
- Shirt before tie
- Shirt before jacket
- Suspenders before jacket
- Belt before jacket
- Tie before jacket
- Underwear before pants

Represent the problem as a graph

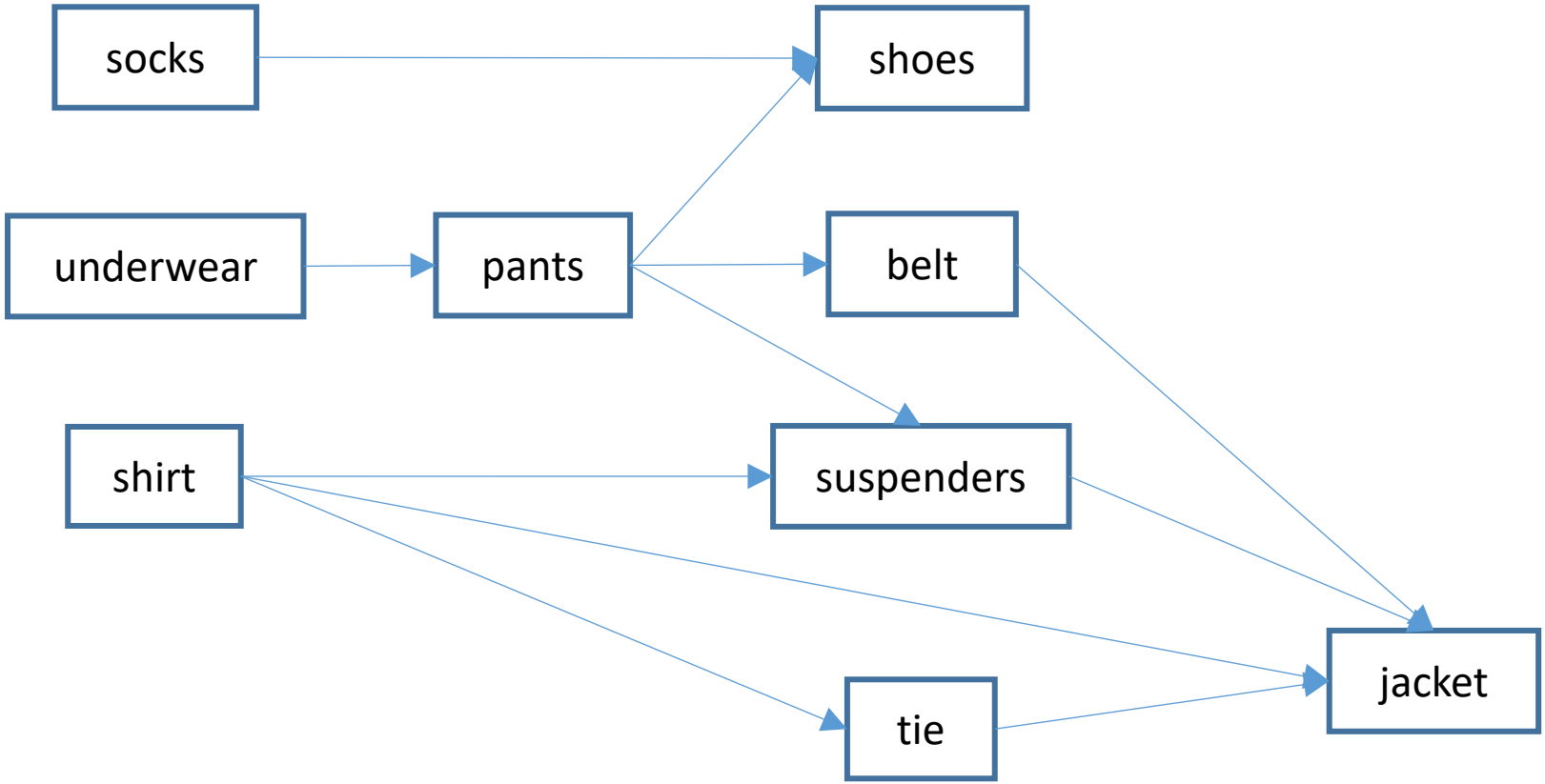
1. V = vertices are the items (tasks)
2. E = edges are the dependencies (constraints) between tasks

- an edge (v → w) means:
  - w is dependent on v, OR (in other words)
  - Task v comes before task w

Clothing graph



Eyeballing a solution



socks … underwear … shirt … pants … shoes … belt … suspenders … tie … jacket

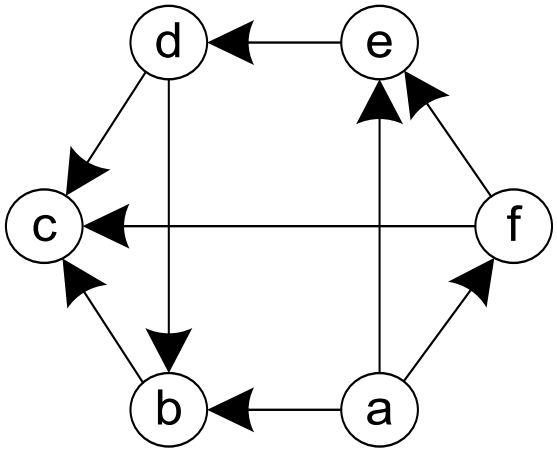
Topological Sort Algo 1: Use Depth First Search

1. Apply DFS to G
   - Starting at any vertex
   - No, really: ANY vertex
2. The order in which vertices become dead ends is the *reverse* of a topological sort order
   - Why?

Example 1

- Assume you have a set of 6 tasks (a, b, c, d, e, f) with the following dependencies:
  - a must be done before b, e, f
  - b must be done before c
  - d must be done before b and c
  - e must be done before d
  - f must be done before c and e

- Step 1: Construct a directed graph to represent the problem (verify it is a DAG)

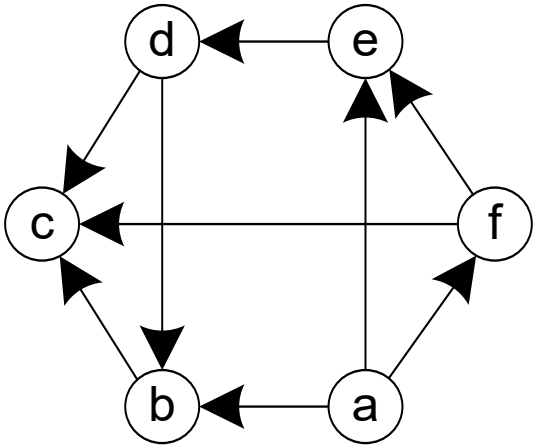


Example 1 (cont)

- Step 2: Apply DFS

  Order vertices become dead ends:

  c b d e f a



- Step 3: Reverse this to get topological sort order:

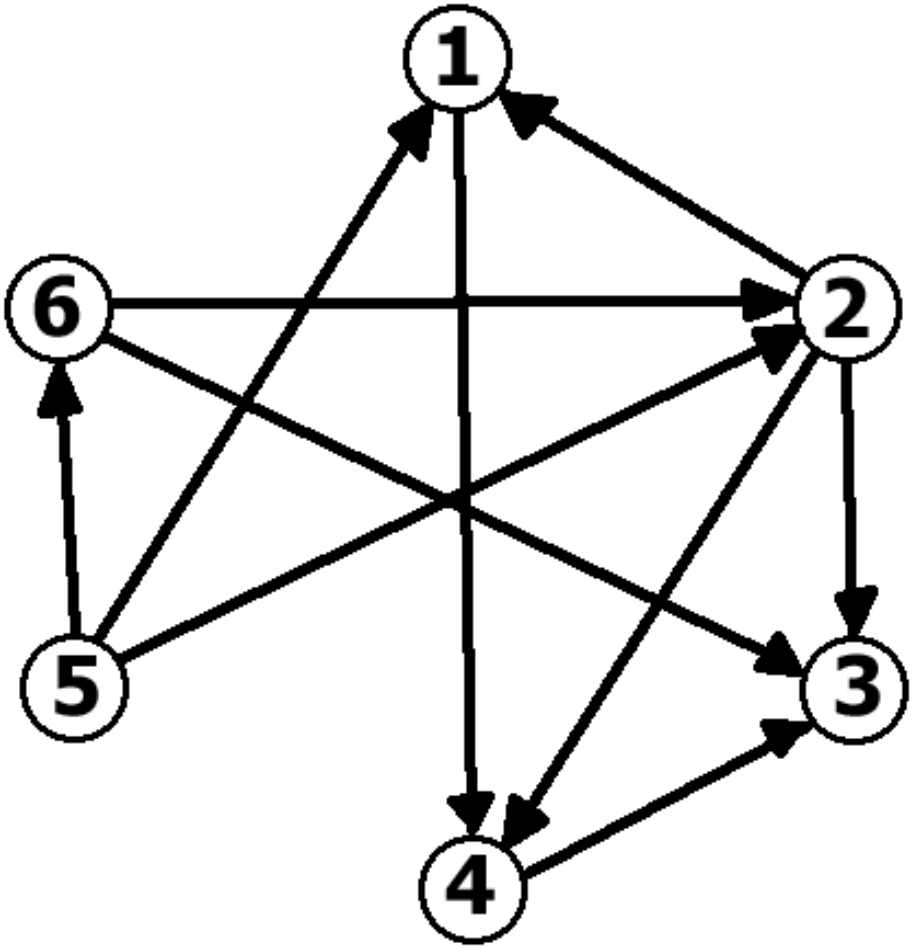
  a f e d b c

Example 2

2 1 (2 before 1)	4 3 (4 before 3)
1 4 (1 before 4)	5 2 (5 before 2)
2 3 (2 before 3)	5 1 (5 before 1)
5 6 (5 before 6)	6 3 (6 before 3)
2 4 (2 before 4)	6 2 (6 before 2)

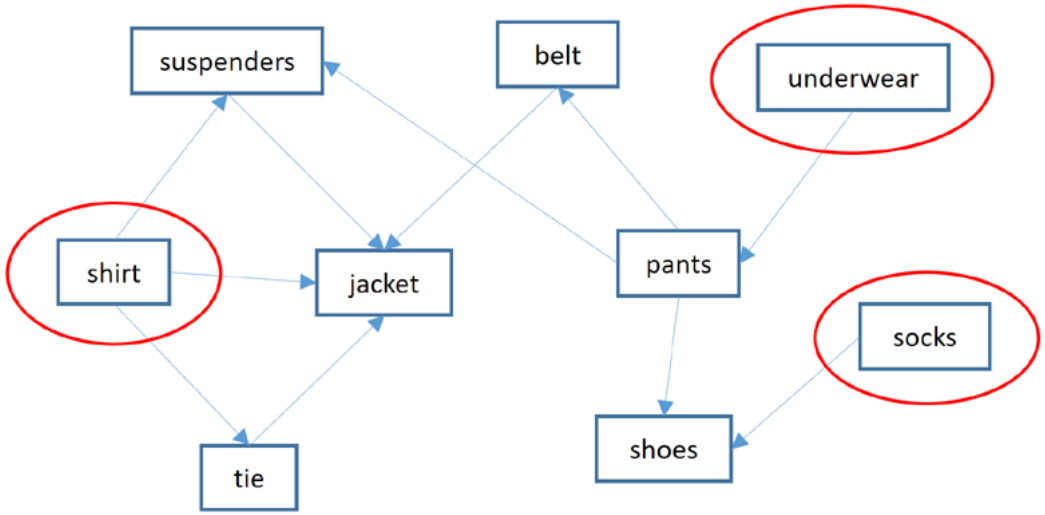
- Step 1: draw the graph (and verify it is a DAG)
- Step 2: apply DFS, get “dead-end” order
- Step 3: reverse this to get topological sort order

Example 2 (cont)



TopoSort Algorithm 2: Decrease (by 1) and conquer

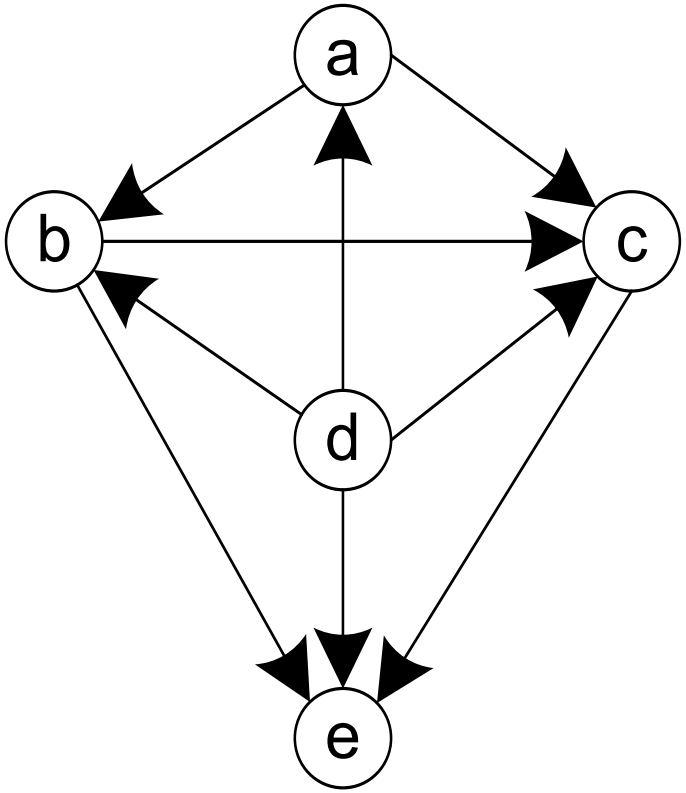
- Key observation:
  - If a vertex v in the dependency graph G has no incoming arrows (*i.e.* in-degree(v) == 0), then v does not have any dependencies
  - It follows that any v that does not have dependencies is a candidate to be visited next in topological order
  - *i.e.* any of these can go first →

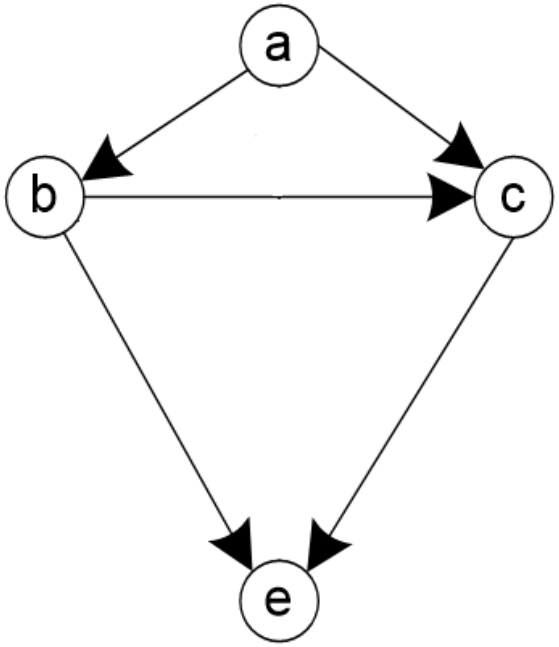


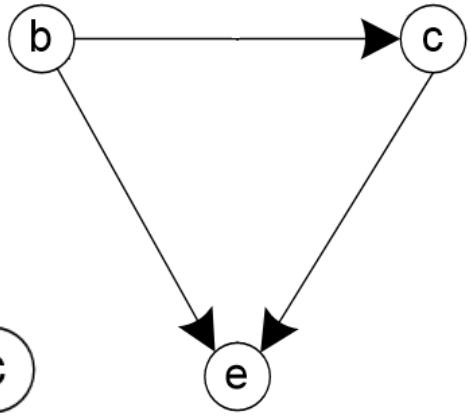
Idea of the algorithm

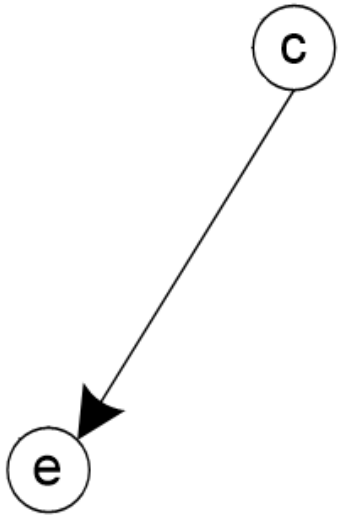
- Identify a $v \in V$ that has in-degree=0
- Delete v and all edges coming out of it
- Repeat until done
- The topological order is the order the vertices are deleted
- If there are $v \in V$, but no v has in-degree=0, the graph G is not a DAG (no feasible solution exists)

Example 3









Algorithm details

- Use a set to store the candidate vertices
  - *I.e.* the vertices with in-degree = 0
  - Any ordered set will do, e.g. TreeSet.
- Use an ordered list to store the delete order
  - Any list type will do, e.g. ArrayList, always adding to the end

TopoSort “Decrease by one” pseudocode

```
Algorithm TopoSort(G)
    create an empty ArrayList A
    create an empty TreeSet Candidates
    add all v with inDegree=0 to Candidates
    while Candidates is not empty
        v = Candidates.first()
        add v to A
        for each vertex w adjacent to v
            remove edge (v,w) from G
            if w has inDegree=0
                add w to Candidates
        remove vertex v from G
    if there are no vertices remaining in G
        solution is in A
    else
        no solution exists
```

Practice problems

- Chapter 4.2, page 142, question 1
- Chapter 5.3, page 185, questions 5 & 6