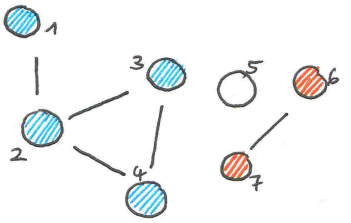
Graphentheorie - DFS

- Previous:
  - graph traversal = 'walk on'/explore graph
  - component = all nodes who can reach each other
  - adjecency list = way to store graph:
    for each node keep list of all neighbours

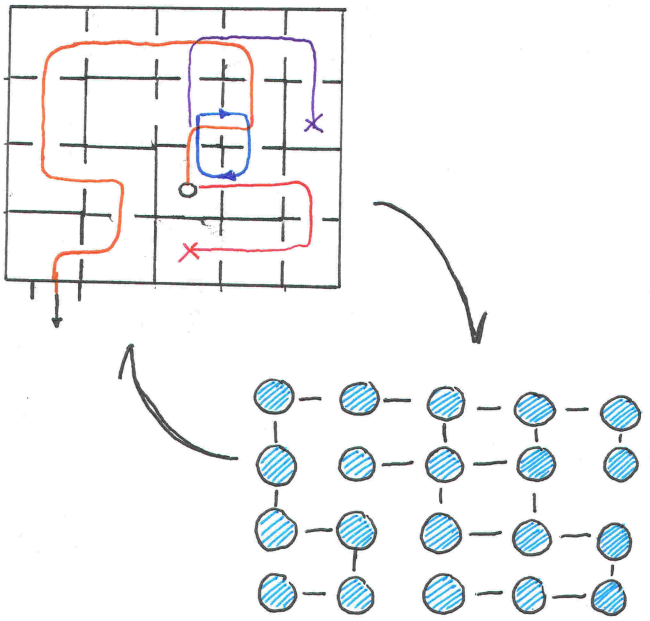
1: 2	5:
2: 1, 3, 4	6: 7
3: 2, 4	7: 6
4: 2, 3	node: neighbours

usually stored as vector of vectors



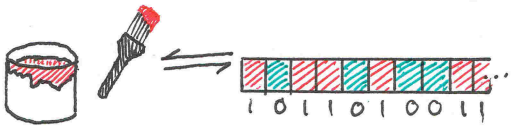
graph with 3 components

- DFS
  - dfs: depth first search
  - goal: traverse entire graph
  - analog Labyrinth
    - are trapped in a labyrinth → find exit
    → view it as a graph:
    rooms = nodes, lanes betw. rooms = edges
  - strategy:
    - fixate a red string in our starting room
    → can always find way back
    - paint a red point in each new room
    → know where we have been
    - visit all neighbours,
      - if neighbour is already marked by red point or has no other neighbours (dead end)
      → return, go back



⇒ always take a new path - repeat unless we discover nothing new.

- DFS Algorithm
  - simulate red points with boolean list
  - red string — function calls
  - In each function call:
    - check if we visited before (return if yes)
    - mark visited
    - visit all neighbours
  - sample Impl. in python




```
visited = [False] * n
def dfs(pos):
    if visited[pos]:
        return
    visited[pos] = True
    for neighbor in graph[pos]:
        dfs(neighbor)
```

.py

- Analysis

Memory: need visited array, one entry per node — $O(n)$

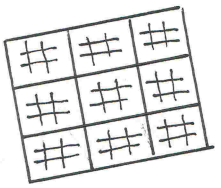
Runtime: visit each node & walk once over every edge — $O(n+m)$

- n = # nodes, m = # edges

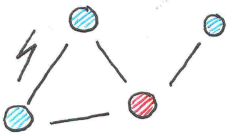
- Applications
  - count components
  - graph coloring
  - finding cycles
  - more advanced: find bridges & articulation points

Graph theory - DFS

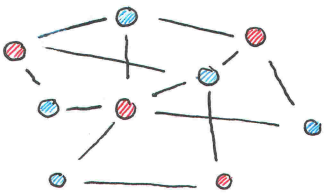
· Application - Graph coloring



- task: want to assign colours to nodes
  but no neighbours shall be the same colours!
  exmpl.: coloring a world map, sodaku, 2 passports, ...
- Sadly very hard problem with many colors
  → can solve for two colors
  check out task tables at prelim.soi.ch



- Idea:
  - fix colour for one node
    → know color for his neighbours
  - continue this for the next neighbour & so on
    & check for a conflict



→ extend DFS
  - give colour as an argument
  - return False if there is a conflict, True if coloring pos.
  - save colour in seperate Array or
    modify visited [0 → unvisited / 1 → visited, blue / 2 → visited, red]