

Algorithms and Data Structures

C2. Graph Exploration: Applications

Gabriele Röger and Patrick Schneider

University of Basel

April 29, 2026

Algorithms and Data Structures

April 29, 2026 — C2. Graph Exploration: Applications

C2.1 Reachability

C2.2 Shortest Paths

C2.3 Acyclic Graphs

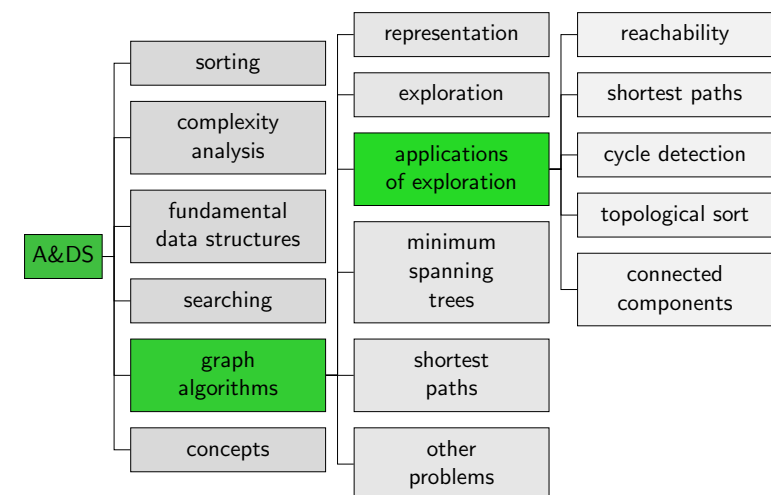
C2.4 Connected Components

C2.5 Summary

Reminder: Graph Exploration

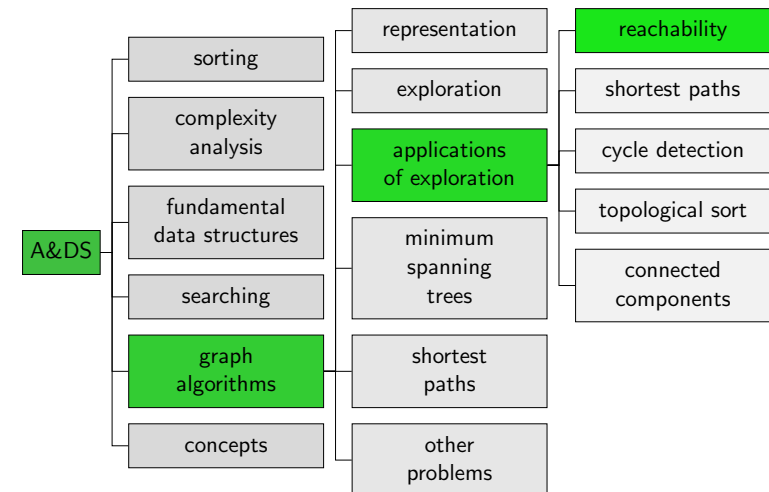
- ▶ Given a vertex v , visit all vertices that are reachable from v .
- ▶ Often used as part of other graph algorithms.
- ▶ **Depth-first search**: go “deep” into the graph (away from v)
- ▶ **Breadth-first search**: first all neighbours, then neighbours of neighbours, ...

Content of the Course



C2.1 Reachability

Content of the Course

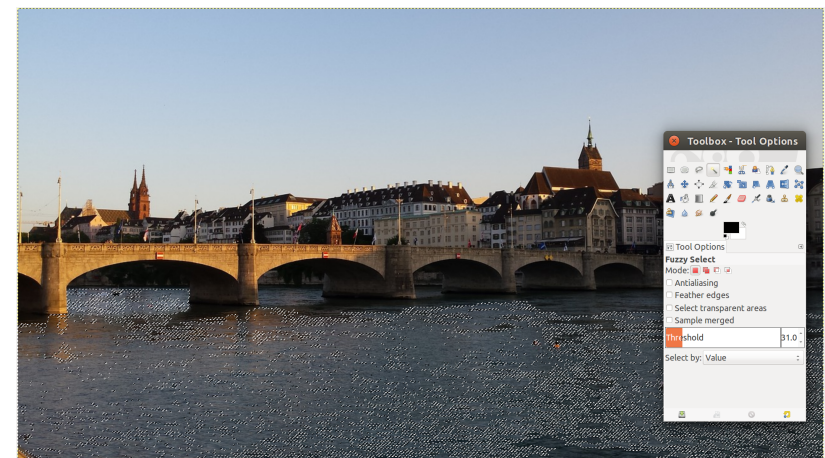


Mark-and-Sweep Garbage Collection

Aim: Release memory occupied by no longer accessible objects.

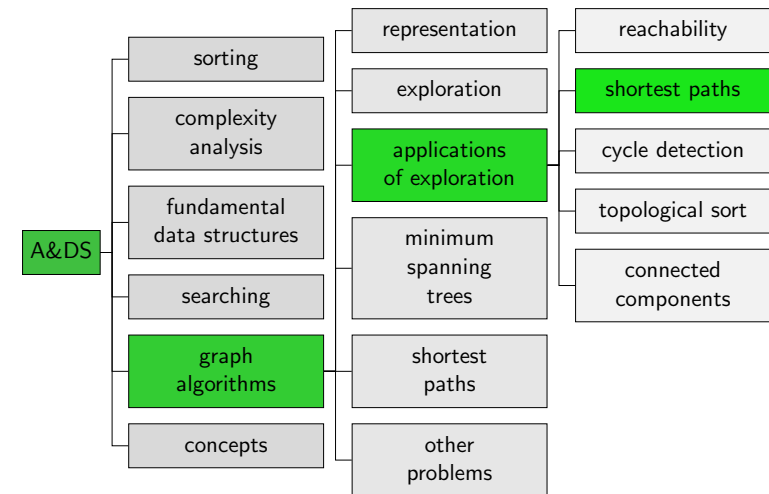
- ▶ Directed graph: **Objects** as vertices, **references to objects** as edges.
- ▶ One bit per object for marker during garbage collection.
- ▶ **Mark:** Mark all reachable objects (set bit to 1).
- ▶ **Sweep:** Clear unmarked objects from memory. Afterwards set bit for all reachable objects back to 0.

Magic Wand in Image Editing



C2.2 Shortest Paths

Content of the Course



Shortest Paths: Idea

- ▶ Breadth-first search visits the vertices with increasing (minimal) distance from the start vertex.
- ▶ First visit of a vertex happens on shortest path.
- ▶ **Idea:** Use path from induced search tree.

Jupyter Notebook



Jupyter notebook: `graph_exploration_applications.ipynb`

Shortest-path Problem

Single-source Shortest-paths Problem

- ▶ Given: Graph and start vertex s
- ▶ Query for vertex v
 - ▶ Is there a path from s to v ?
 - ▶ If yes, what is the shortest path?
- ▶ Abbreviation SSSP

Shortest Paths: Algorithm

```

1 class SingleSourceShortestPaths:
2     def __init__(self, graph, start_node):
3         self.predecessor = [None] * graph.no_nodes()
4         self.predecessor[start_node] = start_node
5
6         # precompute predecessors with breadth-first search with
7         # self.predecessors used for detecting visited nodes
8         queue = deque()
9         queue.append(start_node)
10        while queue:
11            v = queue.popleft()
12            for s in graph.successors(v):
13                if self.predecessor[s] is None:
14                    self.predecessor[s] = v
15                    queue.append(s)
16        ...

```

In principle as before
(just as a class)

Shortest Paths: Algorithm (Continued)

```

19     def has_path_to(self, node):
20         return self.predecessor[node] is not None
21
22     def get_path_to(self, node):
23         if not self.has_path_to(node):
24             return None
25         if self.predecessor[node] == node: # start node
26             return [node]
27         pre = self.predecessor[node]
28         path = self.get_path_to(pre)
29         path.append(node)
30         return path

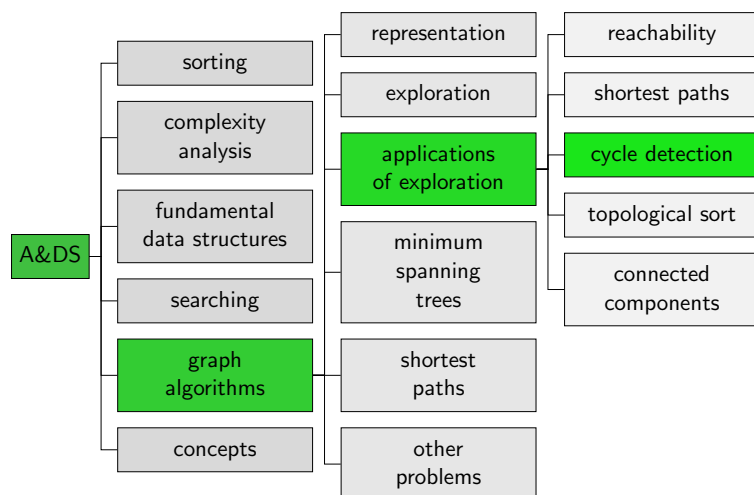
```

Running time?

Later: Shortest paths with edge weights

C2.3 Acyclic Graphs

Content of the Course



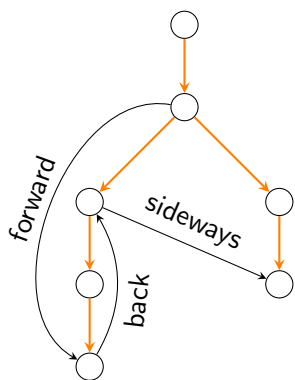
Detection of Acyclic Graphs

Definition (Directed Acyclic Graph)

A **directed acyclic graph** (DAG) is a directed graph that contains no directed cycles.

Task: Decide whether a directed graph contains a cycle. If yes, return a cycle.

Criterion for Acyclicity



Induced search tree of a **depth-first search** (orange) and possible other edges

The (reachable part of the) graph is **acyclic** if and only if there are **no back edges**.

Idea: Remember the vertices on the current path in a DFS.

Cycle Detection: Algorithm

```

1 class DirectedCycle:
2     def __init__(self, graph):
3         self.predecessor = [None] * graph.no_nodes()
4         self.on_current_path = [False] * graph.no_nodes()
5         self.cycle = None
6         for node in range(graph.no_nodes()):
7             if self.has_cycle():
8                 break
9             if self.predecessor[node] is None:
10                self.predecessor[node] = node
11                self.dfs(graph, node)
12
13     def has_cycle(self):
14         return self.cycle is not None

```

Repeated depth-first searches such that at the end all vertices have been visited.

Cycle Detection: Algorithm (Continued)

```

16 def dfs(self, graph, node):
17     self.on_current_path[node] = True
18     for s in graph.successors(node):
19         if self.has_cycle():
20             return
21         if self.on_current_path[s]:
22             self.predecessor[s] = node
23             self.extract_cycle(s)
24             if self.predecessor[s] is None:
25                 self.predecessor[s] = node
26                 self.dfs(graph, s)
27     self.on_current_path[node] = False

```

Found a cycle

Skip if a cycle has been detected somewhere.

Update whether vertex is on the current path.

21 / 40

Cycle Detection: Algorithm (Continued)

When calling `extract_cycle`, `node` is on a cycle in `self.predecessor`.

```

29 def extract_cycle(self, node):
30     self.cycle = deque()
31     current = node
32     self.cycle.appendleft(current)
33     while True:
34         current = self.predecessor[current]
35         self.cycle.appendleft(current)
36         if current == node:
37             return

```

22 / 40

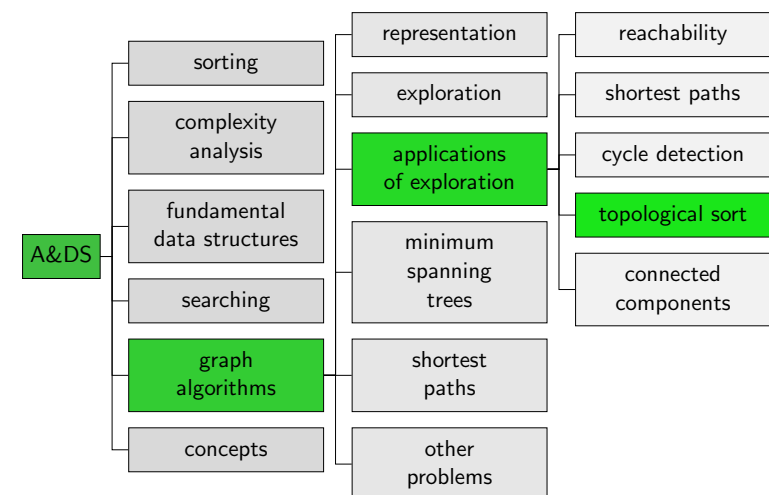
Jupyter Notebook



Jupyter notebook: `graph_exploration_applications.ipynb`

23 / 40

Content of the Course



24 / 40

Topological Sort

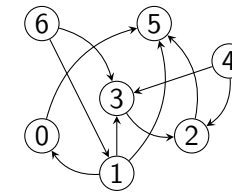
Definition

A **topological sort** of a directed **acyclic** graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v) , then u appears before v in the ordering.

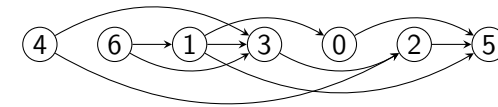
For example relevant for **scheduling**:

edge (u, v) expresses that job u must be completed before job v can be started.

Topological Sort: Illustration



Topological sort: 4, 6, 1, 3, 0, 2, 5



Topological Sort: Algorithm

Theorem

For the reachable part of a acyclic graph, the **reverse DFS postorder** is a topological sort.

Algorithm:

- ▶ Sequence of depth-first searches (for still unvisited vertices) until all vertices visited.
- ▶ Store for each DFS the reverse postorder:
 P_i for i -th search
- ▶ Let k be the number of searches. Then the concatenation P_k, \dots, P_1 is a topological sort.

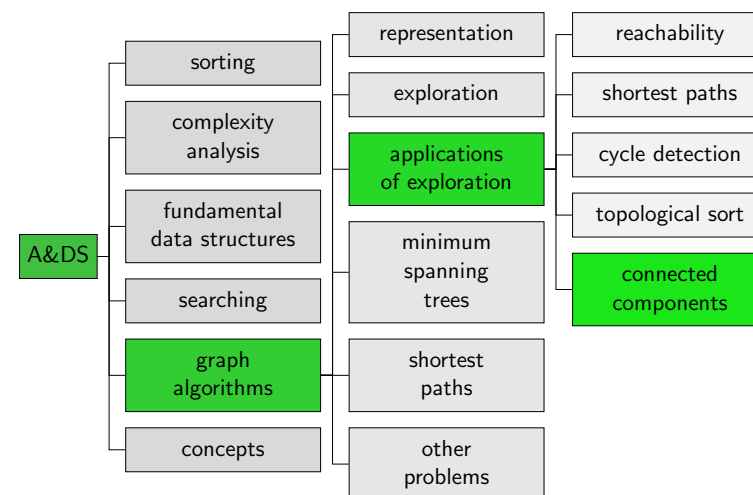
Jupyter Notebook



Jupyter notebook: `graph_exploration_applications.ipynb`

C2.4 Connected Components

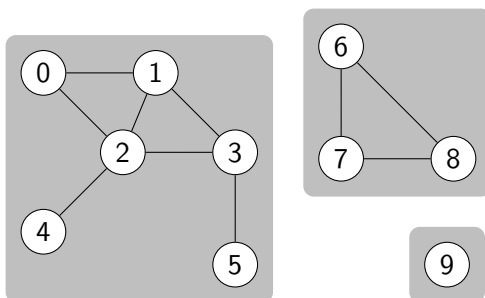
Content of the Course



Connected Components of Undirected Graphs

Undirected graph

- ▶ Two vertices u and v are in the same **connected component** if there is a path between u and v .



Connected Components: Interface

We want to implement the following interface:

```

1 class ConnectedComponents:
2     # Initialization with precomputation
3     def __init__(graph: UndirectedGraph) -> None
4
5     # Are vertices node1 and node2 connected?
6     def connected(node1: int, node2: int) -> bool
7
8     # Number of connected components
9     def count() -> int
10
11     # Component number for node
12     # (between 0 and count()-1)
13     def id(node: int) -> int
  
```

Idea: Sequence of graph explorations until all vertices visited.
ID of vertex corresponds to iteration in which it was visited.

Connected Components: Algorithm

```

1 class ConnectedComponents:
2     def __init__(self, graph):
3         self.id = [None] * graph.no_nodes()
4         self.curr_id = 0
5         visited = [False] * graph.no_nodes()
6         for node in range(graph.no_nodes()):
7             if not visited[node]:
8                 self.dfs(graph, node, visited)
9                 self.curr_id += 1
10
11     def dfs(self, graph, node, visited):
12         if visited[node]:
13             return
14         visited[node] = True
15         self.id[node] = self.curr_id
16         for n in graph.neighbours(node):
17             self.dfs(graph, n, visited)

```

How are connected, count and id implemented?

Jupyter Notebook



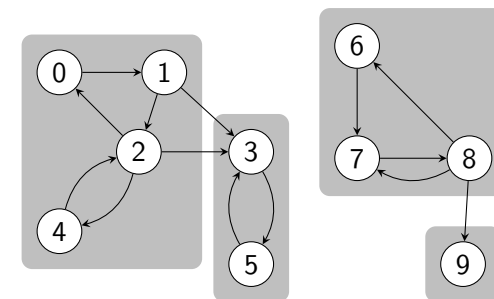
Jupyter notebook: graph_exploration_applications.ipynb

Connected Components of Directed Graphs

Directed graph G

- ▶ If one ignores the arc directions, then every connected component of the resulting undirected graph is a **weakly connected component** of G .
- ▶ G is **strongly connected**, if there is a directed path from each vertex to each other vertex.
- ▶ A **strongly connected component** of G is a maximal strongly connected subgraph.

Strongly Connected Components



Strongly Connected Components

Kosaraju' algorithm

- ▶ Given directed graph $G = (V, E)$, compute a reverse postorder P (for all vertices) of the graph $G^R = (V, \{(v, u) \mid (u, v) \in E\})$ (all edges reversed).
- ▶ Conduct a sequence of explorations in G , always selecting the first still unvisited vertex in P as the next start vertex.
- ▶ All vertices that are reached by the same exploration, are in the same strongly connected component.

Jupyter Notebook



Jupyter notebook: `graph_exploration_applications.ipynb`

C2.5 Summary

Summary

We have seen a number of applications of graph exploration:

- ▶ Reachability
- ▶ Shortest paths
- ▶ Cycle detection
- ▶ Topological sort
- ▶ Connected components

Some applications require a specific exploration, for other applications we can use both, BFS and DFS.