Depth-first search
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Path-finding

One of the most basic problems in graph theory: Given a graph G and two vertices $x, y \in V(G)$, is there a path from x to y?

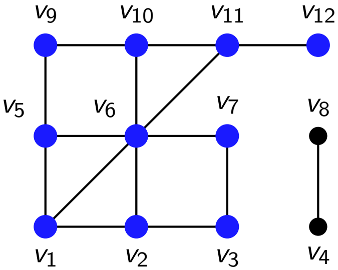
E.g. can an enemy attack the base without breaking down a wall?

Often we want to know the **shortest** path from x to y — see next video!

Component-finding

In fact, it's better to ask for something more.

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.



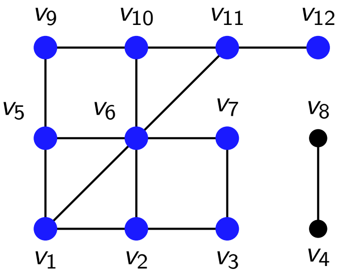
Input: v_1

Output: $[v_1, v_2, v_3, v_5, v_6, v_7, v_9, v_{10}, v_{11}, v_{12}]$

Component-finding

In fact, it's better to ask for something more.

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.



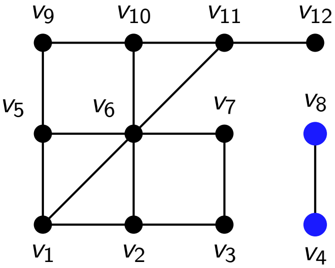
Input: v_6

Output: $[v_1, v_2, v_3, v_5, v_6, v_7, v_9, v_{10}, v_{11}, v_{12}]$

Component-finding

In fact, it's better to ask for something more.

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.



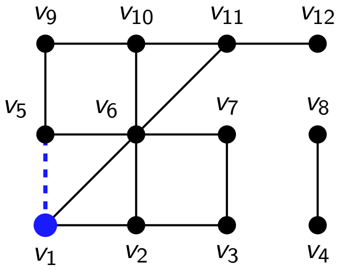
Input: v_4
Output: $[v_4, v_8]$

In other words, we check whether there is a path from x to y for **all** y.
Turns out the worst-case running time is the same either way!

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.

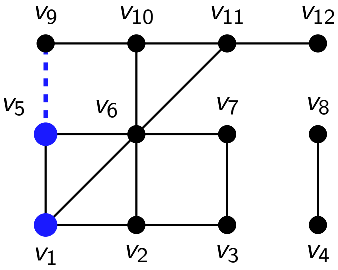


Input: G, v_1
Output: $[v_1$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.

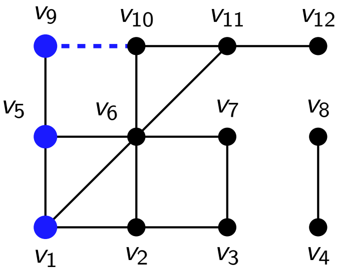


Input: G, v_1
Output: $[v_1, v_5$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



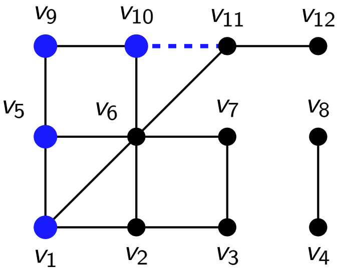
Input: G, v_1

Output: $[v_1, v_5, v_9$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

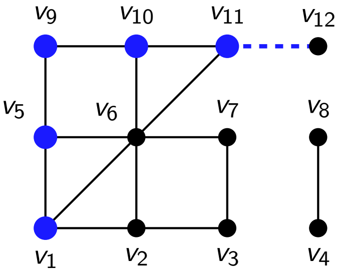
Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



Input: G, v_1
Output: $[v_1, v_5, v_9, v_{10}$

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.

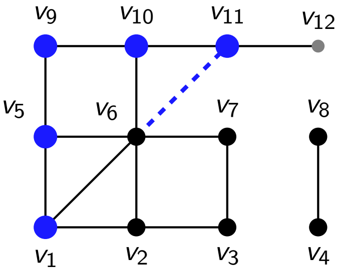


Input: G, v_1
Output: $[v_1, v_5, v_9, v_{10}, v_{11}$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



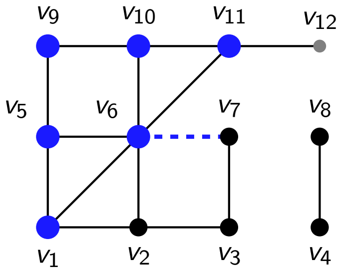
Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



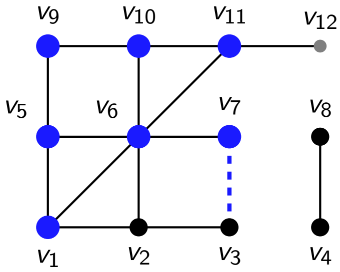
Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}, v_6$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



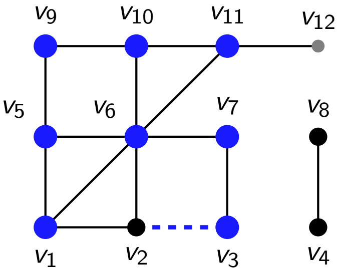
Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}, v_6, v_7$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.

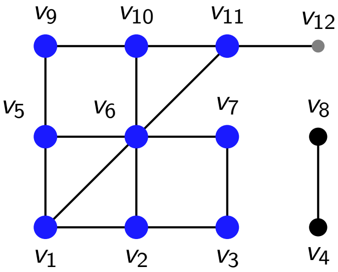


Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}, v_6, v_7, v_3$

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.



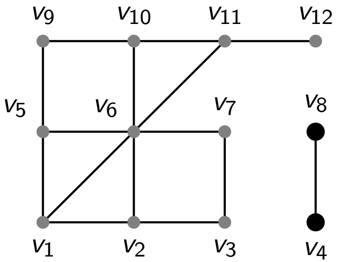
Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}, v_6, v_7, v_3, v_2$

Depth-first search: The idea

Input: A graph G and a vertex $x \in V(G)$.
Output: A list of all vertices in the component of G containing x.

Idea: Think of the graph as like a **maze**: explore greedily until everything looks familiar, then backtrack.

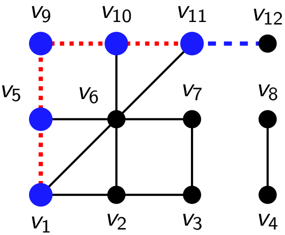


Input: G, v_1

Output: $[v_1, v_5, v_9, v_{10}, v_{11}, v_{12}, v_6, v_7, v_3, v_2]$

The slick way to implement this is to use recursion.

Pseudocode and example



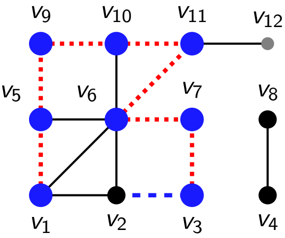
Input: v_1

Algorithm: DFS

Input : Graph $G = (V, E)$, vertex $v \in V$.
Output : List of vertices in v's component.

Number the vertices of G as $v_1, \dots, v_n$.
Let explored$[i] \leftarrow 0$ for all $i \in [n]$.
Procedure `helper(`v_i`)`
  if explored$[i] = 0$ **then**
    Set explored$[i] \leftarrow 1$.
    for v_j *adjacent to* v_i **do**
      if explored$[j] = 0$ **then**
        Call `helper(`v_j`)`.
Call `helper(`v`)`.
Return [v_i : `explored`$[i] = 1$] (in some order).

Pseudocode and example



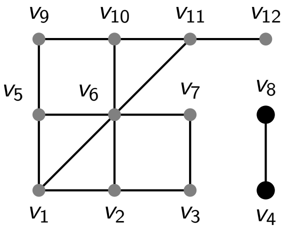
Input: v_1

Algorithm: DFS

Input : Graph $G = (V, E)$, vertex $v \in V$.
Output : List of vertices in v's component.

Number the vertices of G as $v_1, \ldots, v_n$.
Let explored$[i] \leftarrow 0$ for all $i \in [n]$.
Procedure `helper(`v_i`)`
  if explored$[i] = 0$ **then**
    Set explored$[i] \leftarrow 1$.
    for v_j *adjacent to* v_i **do**
      if explored$[j] = 0$ **then**
        Call `helper(`v_j`)`.
Call `helper(`v`)`.
Return $[v_i :$ `explored`$[i] = 1]$ (in some order).

Pseudocode and example



Input: v_1

Algorithm: DFS

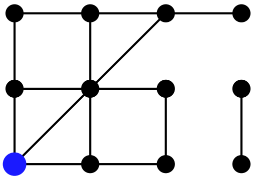
Input : Graph $G = (V, E)$, vertex $v \in V$.
Output : List of vertices in v's component.

1 Number the vertices of G as $v_1, \dots, v_n$.
2 Let explored$[i] \leftarrow 0$ for all $i \in [n]$.
3 **Procedure** `helper(`v_i`)`
4   **if** explored$[i] = 0$ **then**
5   Set explored$[i] \leftarrow 1$.
6   **for** v_j *adjacent to* v_i **do**
7   **if** explored$[j] = 0$ **then**
8   Call `helper(`v_j`)`.
9 Call `helper(`v`)`.
10 Return $[v_i : \texttt{explored}[i] = 1]$ (in some order).

We assume G is in adjacency list form.

Time analysis: In total there are $\sum_{v \in V} d(v) = O(|E|)$ calls to `helper` (each vertex only runs lines 5–7 once), and there is $O(1)$ time between calls. So the running time is $O(|V| + |E|)$.

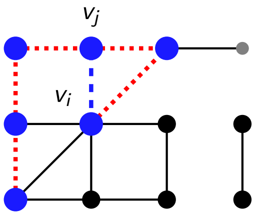
Correctness I: Output is contained in v's component C



Invariant: "When `helper` is called, if `explored`$[i] = 1$ then $v_i \in V(C)$."

Proof by induction. Vacuously true for initial call and second call. ✓

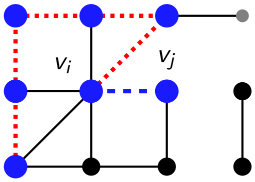
Correctness I: Output is contained in v's component C



Invariant: "When `helper` is called, if `explored`$[i] = 1$ then $v_i \in V(C)$."

Proof by induction. Vacuously true for initial call and second call. ✓

Suppose it holds at the start of some call `helper`(v_j) from `helper`(v_i).
If v_j is already explored, we're done.



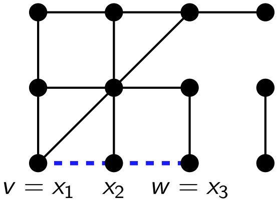
Invariant: "When helper is called, if explored$[i] = 1$ then $v_i \in V(C)$."

Proof by induction. Vacuously true for initial call and second call. ✓

Suppose it holds at the start of some call helper(v_j) from helper(v_i).
If v_j is already explored, we're done. If not, we must show $v_j \in V(C)$.

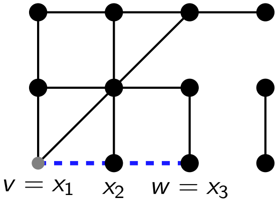
Since we called from helper(v_i), $\{v_i, v_j\} \in E$ and v_i is explored.
By induction there is a path P from v to v_i. Then Pv_iv_j is a walk from v to v_j, which contains a path, so $v_j \in V(C)$. □

Correctness II: Output contains v's component C



Let $w \in V(C)$. Then there is a path $P = x_1 \dots x_t$ from v to w.

Correctness II: Output contains v's component C



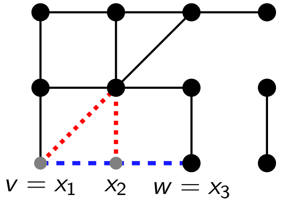
Let $w \in V(C)$. Then there is a path $P = x_1 \ldots x_t$ from v to w.

Claim: Every vertex in P is explored.

Proof by induction: We prove $x_1, \ldots, x_i$ are explored for all $i \leq t$.

x_1 is explored. ✓

Correctness II: Output contains v's component C



Let $w \in V(C)$. Then there is a path $P = x_1 \dots x_t$ from v to w.

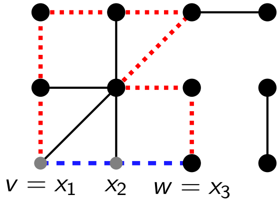
Claim: Every vertex in P is explored.

Proof by induction: We prove $x_1, \dots, x_i$ are explored for all $i \leq t$.

x_1 is explored. ✓

If x_i is explored, then `helper`(x_{i+1}) will be called from `helper`(x_i), so x_{i+1} will also be explored (either then or earlier).

Correctness II: Output contains v's component C



Let $w \in V(C)$. Then there is a path $P = x_1 \dots x_t$ from v to w.

Claim: Every vertex in P is explored.

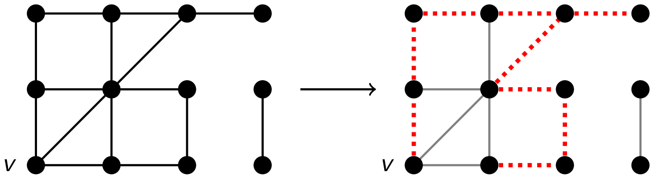
Proof by induction: We prove $x_1, \dots, x_i$ are explored for all $i \leq t$.

x_1 is explored. ✓

If x_i is explored, then helper(x_{i+1}) will be called from helper(x_i), so x_{i+1} will also be explored (either then or earlier). □

Depth-first search trees

Consider the subgraph formed by the edges **traversed** in DFS:



This is an example of a **DFS tree** rooted at v.

Definition: A **DFS tree** T of G is a rooted tree satisfying:

- $V(T)$ is the vertex set of a component of G;
- If $\{x, y\} \in E(G)$, then x is an ancestor of y in T or vice versa.

Theorem: DFS always gives a DFS tree. (See problem sheet.)

DFS trees can be independently useful! (See problem sheet.)

Depth-first search works for directed graphs too, in exactly the same way. But paths **between** v and w are replaced by paths **from** v **to** w.